Stateful Distributed Firewalls

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by

T. W. Verwoerd

University of Canterbury
2001

Abstract

A fundamental feature in current network security architectures is the monolithic firewall. This thesis presents an alternative design, consisting of a cluster of small firewall nodes, that offers better economy, scalability, failure recovery and potentially, greatly increased processing power. These improvements allow the use of computationally expensive firewalling and IDS techniques to offer effective protection against all types of network attack. Also presented are techniques for developing fault−tolerant proxy applications, maintaining connections in spite of node failures, and a novel load balancing design. Generic Load Balancing (GLOB) uses per−node filtering to distribute network load transparently in a cluster without any single points of failure. This thesis also presents evaluations of prototype implementations of these techniques.

Acknowledgements

There are a few people I'd like to take this opportunity to thank. First is a God who never fails me. Second are my parents: I have always been able to count on their support and advice. Third is my supervisor − Ray Hunt. Finally, there are all the others, each with their own contribution: Diana, Del, Michael − thank you for giving me a better perspecitive.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Security in the new Internet

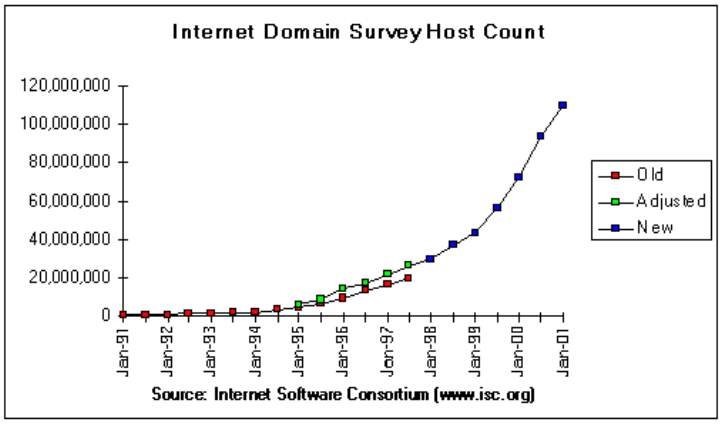

Figure 1. Estimated number of hosts on the Internet [ISC]

Internetworking, and in particular the Internet, is fundamental to many private and commercial systems. The use of email, web access, and electronic commerce is fast becoming as ubiqutous as the telephone. In recent years networks have grown exponentially, with recent estimates placing the total number of connected hosts as high as 109 [ISC] (January 2001) or 122 million [Netsizer] (May 2001). Netcraft's April 2001 Web Server Survey [Netcraft] inlcudes more than 28.5 million sites.

The Internet is really big.

With the recent introduction of broadband end−user technologies (such as cable modems, sattelite links and xDSL [Broadband]), private systems are becoming permanently connected to the Internet. As a resul, the bandwidth demand for popular services is rocketing. In addition, providing services (such as web servers) from private systems has become more feasible, leading to innovations such as dynamic DNS [DynDNS].

Low−end users can get fast, persistent connections.

Systems such as these typically belong to private individuals or small organisations, that do not have the technical or economical resources to ensure security for their systems. As a result, even script−kiddy hackers (see Section 3.1) can assemble fleets of compromised systems [GRCHack]. Even in cases where security measures are in place, the low−end devices and lack of maintenance lead to security problems.

Low−end users have little or no security.

On the other hand, these developments allow access to e−commerce to small

businesses, allowing these to compete effectively with established companies. eMarketer's "eCommerce: B2C Report" notes that online spending was US$59.7 in 2000 − up 98% from 1999, with similar future growth expected [eCommerce]. In order to share in this market sector, however, any business needs to maintain the trust of its customers − which means implementing strong security.

E−commerce is lucrative to small businesses.

Often, e−commerce security centres around cryptographic techniques for transferring sensitive information between the client and server, typically HTTPS. As vividly demonstrated by a series of breakins to more than 40 e−banking and e−commerce sites [CCHack], where over a million credit cards' details were exposed, network security is as important, if not so, as link security.

Network security is critical to securing e−commerce.

As demonstrated in the 2001 CSI/FBI Computer Crime and Security Survey [CSI], and the PCWeek hackers' challenge [HTTPSHack], real network security requires more than just a firewall and intrusion detection system. The CSI respondents have a range of implemented security controls − 92% have physical security, 95% have firewalls, 90% access control and 61% intrusion detection − yet they also list over US$377 million in computer−related losses. This figure covers only 37% of respondents − the remainder of the 78% that acknowledge losses couldn't quantify a dollar value.

Attacks succeed in spite of technical security measures.

Typical modern security comprises some combination of intrusion detection, firewalls and hardened hosts. As discussed in Section 6, these techniques are ineffective in preventing intrusions − instrusion detection does not prevent attack, merely reporting the fact, firewalls do not block application−level attacks, and hardened hosts must be constantly maintained.

Current security techniques cannot protect against current attack techniques.

Legal remedies are also often ineffective. Prosecuting offenders is expensive (US$200,000 for an FBI prosecution according to [GRCHack]), technically difficult [Sommer−1, Sommer−2], and sometimes infeasible − when the offender is in a country without suitable laws [Bellovin] or young [GRCHack].

The legal system is not an effective remedy or deterrent.

Therefore, there is a real need for a security architecture that is cheap, low

maintenance, and is capable of actively protecting network systems. In addition, such a system must be capable of handling relatively large network bandwidths. This thesis presents the first step towards such an architecture − a framework for creating cheap, high−performance firewalls.

1.2. The next step in network security

As described in the previous section, current network security techniques are lagging behind the needs of modern networks. These techniques, taken together, have the ability to protect networks effectively, but only when applied as a single unit.

Firewalls are an effective form of network access control. They have been successful in eliminating most network−level attack techniques, and in protecting internal services. Current firewalls are generally all−or−nothing systems − services are either protected completely, or not. In particular, firewalls are ineffective in protecting public services against application−level attacks.

Intrusion detection systems can recognise many forms of application−level attack. They offer detailed reporting on what attacks took place, typically after the fact − due to their placement in parallel to the recipients. Processing network captures for attack recognition is computationally intensive, due to the volume of data involved and the protocol−specific tracking required.

Combining these two techniques, creating a hybrid firewall and IDS system, is not an original concept. This approach has two major difficulties − the complexity of the resulting system, and the processing requirements. The first can be addressed to a large extent by the use of firewall proxies − allowing firewall−style control of application data. The second is more difficult to address.

This thesis proposes a combination of a load balancing system designed to support gateway systems such as firewalls, and application proxies with failover support as a solution to these problems. The use of a distributed platform allows the use of cheap, low−end computers while offering high−end processing power, potentially exceeding that offered by monolithic systems. Application proxies offer extensive flexibility in protocol handling, including checking connections for attacks − allowing the firewall to prevent any level of known attack.

1.3. Background sections

Section 2 provides an overview of the TCP/IP protocol suite. Readers may wish to

skim this section to reaquant themselves with the details of these protocols, or may skip directly to later sections. This section also includes notes on the interaction between TCP/IP protocols and security architectures, in particular the distributed firewall system presented here.

Section 3 presents an overview of network attacks − the sources of attacks and the techniques used. A reader that is familiar with the dominance and effectiveness and techniques involved in application−level attacks may choose to just glance through this.

Sections 5, 9 and 10 describe existing firewalling, load balancing and high availability techniques. These provide background and context for the original work presented here.

1.4. Original work

Sections 4 and 6 outline the case and design for a distributed firewall using a cluster of gateways. In these sections, we explain the requirements for such a cluster system (consisting of a number of cooperating gateway nodes), and how this approach complements the shortcomings of current security techniques.

Section 7 describes the design of a transparent proxy framework that supports failover without dropping connections. The source code of two implementations based on this framework − for protocol−neutral and HTTP proxies − is presented in Appendix 3.

Section 8 goes into more detail on our connection hijacking component, *prism*, used to recreate TCP connection endpoints lost during node failures.

Sections 11 and 12 describe our load balancing prototype of a Generic LOad−Balancing (GLOB) system. Section 11 presents a technique for configuring nodes to offer simple, failure−tolerant distribution of network traffic to cluster nodes. Section 12 builds on this, presenting a filtering framework that eliminates packet duplication, effectively sharing the presented load among nodes.

Section 13 concludes this thesis, with a brief summary of what has been achieved, and what remains to be done. This is followed by Section 14, the references, and Section 15, a glossary.

Finally, the Appendices (Section 15 and up) present details on ideas related to this thesis not yet implemented, and source code for the components described in previous sections.

2. Foundations of the Internet: TCP/IP

The modern−day Internet runs predominantly on the protocol suite collectively known as TCP/IP, after two of its most important members − the Transport Control Protocol and the Internet Protocol. This suite[1] offers host−to−host addressing and best−effort delivery (IP, UDP), reliable ordered byte streams (TCP), error notification (ICMP), and numerous application protocols: the world wide web (HTTP), file transfer (FTP), email (SMTP, POP, IMAP), device monitoring (SNMP) and symbolic name resolution (DNS).[2]

In the development of our firewall, we will be focusing on TCP/IP−based protocols almost exclusively. This is in part due to the dominant position of these protocols in the market − every system connected to the Internet uses some form of TCP/IP, which is where firewalls find their greatest use. Secondly, the issues and challenges present in TCP/IP also apply to many other protocols − this suite offers a rich variety of different requirements and behaviours.

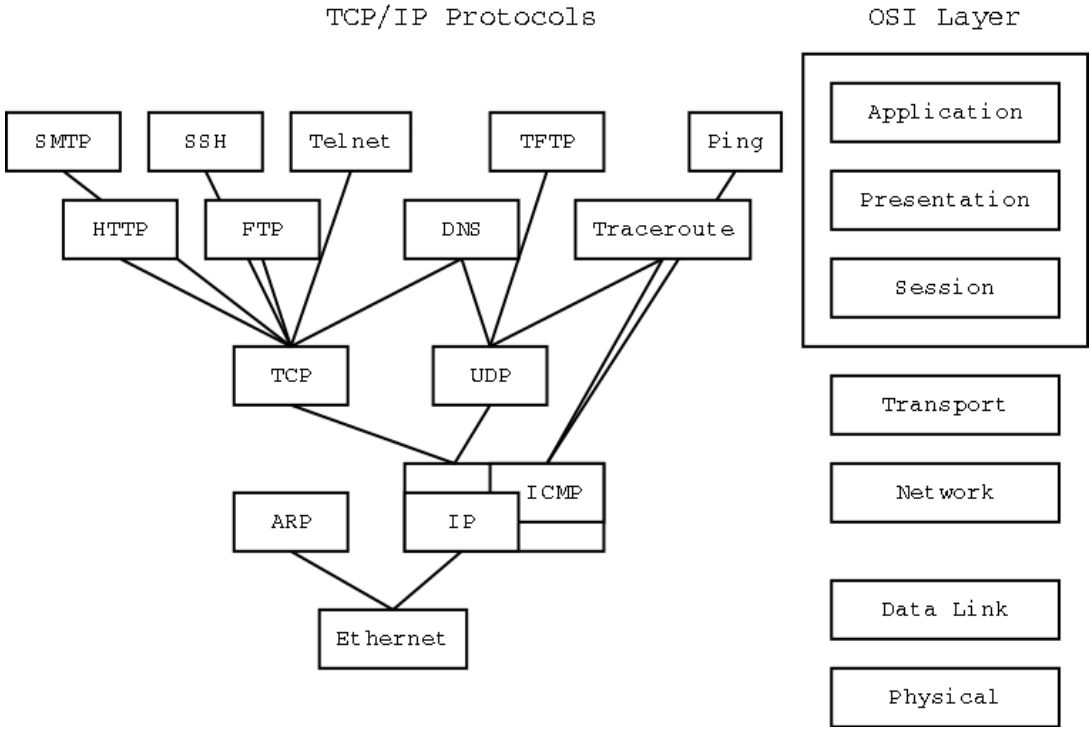

Figure 2. The OSI Protocol Layering Model and part of the TCP/IP Protocol Suite

To place the various protocols discussed here into context, consider Figure 2. It shows the OSI protocol layers, and the matching TCP/IP−based protocols, as well as

1 Refer to [IPStandards] for a complete list of Internet standards and protocols.

2 In this thesis, we will be focusing on IP version 4, the protocol suite in general use. Many of the same principles apply to the IP version 6 [IPv6] currently being introduced.

dependencies between protocols.

2.1. Internet Protocol (RFC791)

The Internet Protocol (IP) [IP] is the basis of the Internet and the TCP/IP suite. It has a number of notable aspects:

- Delivery is host−to−host: each device is identified by a unique IP address, to which any packets for that device are routed.
- IP is a datagram service: data is segmented into discrete packets, each of which contains sufficient header information to allow delivery of that packet.
- IP is connectionless: there is no connection setup or teardown. Any packet is routed according to the current network state at the time of transfer.
- IP is unreliable: when an intermediary node has insufficient resources to process a packet, it is simply discarded − IP allows, but does not require, an error notification to be returned. Protocols built upon IP must compensate for this if necessary.
- IP routes packets in a next−hop fashion: each intermediary node forwards packets to its best guess at the optimal next waypoint. Incorrect routing tables can result in out−of−order packet delivery or packet looping.
- IP offers packet fragmentation, allowing large packets to be transferred over connections that limit packet sizes. Packet fragmentation is currently depreciated for TCP, which has a path MTU (maximum transmission unit) discovery mechanism.
- IP has no inherent flow control: higher layer protocols must handle network congestion and under−utilisation directly.

Higher−level data is encapsulated in IP packets by prefixing with the header shown in Table 1.

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	Version		Header Length		Type of Service (ToS)				Total Length (in bytes)							
4	Identification								Flags			Fragment Offset				
									0	DF	MF					
8	Time to Live (TTL)				Protocol				Header Checksum							
12	Source IP Address															
16	Destination IP Address															
20..	Options (if present)															
	Data															

Table 1. IPv4 packet header format[3]

The fields in the IP Packet Header are:

- *Version* − This thesis uses IPv4 as a basis, so this field has a constant value of four.
- *Header Length* − The length of the IP header in 32−bit words − five if no options are present.
- *Type of Service* − A bitmask field containing information on service constraints and precedence of packets. In theory, ToS values would allow IP packets to be routed or prioritised based on the traffic carried − in practice, few applications use this facility in current implementations.[4]
- *Total Length* − The total length of the datagram in bytes (up to 65535 bytes). In general, however, packet sizes are constrained by the underlying protocol. Unless some path MTU mechanism is used, IP fragmentation is used for oversize packets.
- *Identification* − A number that uniquely identifies an IP datagram for use in packet defragmentation.
- *Flags* − Contains the DF (Don't Fragment) and MF (More Fragments) flags; used to control datagram fragmentation.
- *Fragment offset* −Offset of this fragment from the beginning of the datagram, in units of 8 bytes.
- *Time to Live (TTL)* − Indicates an upper limit to the number of routers a packet may traverse. Each router (gateway) that a packet traverses decrements this value by one (or more), and the packet is discarded when this value reaches zero.
- *Protocol* − A numerical field indicating which protocol is encapsulated in this

3 The header and leftmost columns indicate bits (across) and bytes (down), respectively.
4 Refer to [AssgnNo] and [ToS/DS] for more detail.

packet. Common values are shown in Table 2. [AssgnNo]

Protocol	***Numerical Value***
Internet Control Message Protocol (ICMP)	1
IP−in−IP encapsulation (IP)	4
Transmission Control Protocol (TCP)	6
User Datagram Protocol (UDP)	17

Table 2. Selected IP Encapsulated Protocol Values

- *Checksum* − A 16−bit one's complement sum of the IP header, used to detect packet corruption in transit. No error message is generated on a corrupted packet discard.
- *Source IP Address* − The IP address of the host interface that generated this packet. See Section 2.1.4 for more detail on IP addresses.
- *Destination IP Address* − The IP address of the host interface to which this packet is directed, or the IP address associated with a broadcast or multicast group.
- *IP Options* − An IP header may, optionally, include special fields to modify packet handling on the IP level (padded out to 32−bit boundaries if needed). Currently available options include:
  - *Security:* Used to send packet security information (military use).
  - *Loose Source Routing:* Specifies nodes that the packet must traverse en route, but allows arbitrary intermediary nodes.
  - *Strict Source Routing:* Specifies nodes that the packet must traverse en route − the packet is restricted to only traverse those specific nodes.
  - *Record Route:* Allows the packet to record which nodes (up to the maximum header length) have been traversed en route.
  - *Internet Timestamp:* Similar to *Record Route*, but each entry is also branded with the router time (GMT) in milliseconds.
- *Data* − The datagram payload, identified by the *Protocol* field.

In particular, three issues require greater explanation:

2.1.1. IP Packet Fragmentation

When an IP packet is routed into a link that has a maximum packet size smaller than that of the packet, one of two things happens:

1. If the DF flag is set, the packet is discarded. An ICMP Destination Unreachable: Fragmentation Required message is sent back to the packet source − this behaviour

is used in TCP Path Maximum Transmission Unit (MTU) discovery. When TCP receives such an error response, it reduces the IP packet size (increasing it when no error is returned). This avoids the alternative result: packet fragmentation.

2. If DF is not set, the IP packet is fragmented. The packet payload is broken up into a number of smaller blocks, each with its own full IP header. Every fragment has the same IP packet ID, the offset of this payload segment into the original set as Fragment Offset, and the MF flag set (except for the last fragment). These fragments are then forwarded onwards, and can be refragmented further if necessary.

Fragmentation adversely affects performance, as the end host has to buffer and reassemble the full IP packet before handing it to the higher layers. If any fragment is lost, the entire packet has to be retransmitted.

A more pertinent issue in our context is the security implications introduced by IP fragmentations [FragSec]. Firewalling schemes that rely on higher−level protocol headers (notably TCP and UDP) will be unable to filter non−first fragments; the header information is not present in these packets. One solution is to only filter initial fragments − by blocking the first fragment, packet reassembly is prevented leading to the packet later being discarded.

This approach can lead to a Denial of Service (DoS) attack against firewalled hosts: by flooding a host with isolated (non−first) fragments, that host is forced to buffer these until a timer expires. Since each fragment can indicate to a packet of up to 64Kb in size, this can lead to resource problems on the host. Another method is to send illegally fragmented packets (such as 0−length fragments [0−Frag] or fragments resulting in oversize packets [PingOfDeath]) that may crash host systems.

Different systems handle (illegal) overlapping fragments in different ways. By taking advantage of this fact, an attacker can use overlapping fragments to hide information from packet filtering and IDS systems [Ptachek]. Consider a firewall consisting of a packet filter that only allows packets to or from TCP port 80, and filters only initial fragments. This firewall protects a server (such Solaris or Windows NT) that handles fragment overlaps by using the old data.

IP Header: MF unset, Fragment Offset = 2	
	TCP Destination Port = 25
... Rest of TCP header and data	

IP Header: MF Set, Fragment Offset = 0	
TCP Source Port = 1059	TCP Destination Port = 80
... Rest of TCP header and garbage data	

IP Header: MF unset, Fragment Offset = 0	
TCP Source Port = 1059	TCP Destination Port = 25
... Rest of TCP header and data	

Table 3. Using fragmentation to bypass packet filtering: fragmented packets, and the result of reassembly

First, an attacker sends the second (and final) fragment of an IP packet, set with an offset of 2, and a first data word (16 bits) of 25 − aligned to be the destination port field of the TCP header (see Table 3). The packet filter does not handle non−first fragments, so this is passed on to the host and is buffered. Next, the attacker sends an initial fragment with a TCP header destination port set to 80 − which is passed through by the packet filter.

At the host, the older data takes precedence, so the destination port of the reassembled packet is 25 − and the packet is passed to a protected (SMTP) port. The attacker has successfully bypassed the packet filter, and can use the same technique to bypass naive ID systems.

A solution used in the Linux kernel, and hence in our model firewall, is to allow the firewall to defragment packets. While this has performance implications for the firewall (similar to the DoS scenario described above), this protects internal systems, including IDS monitoring internal networks, from fragment−based attacks through the firewall.

2.1.2. IP Tunnelling

The use of tunnelling protocols[5] over IP can complicate mechanisms based on the IP packet header, since the standard IP header may have little relation to the

5 Tunnelling protocols encapsulate low−level protocol data units as payload to other protocols (for example, IP−over−IP), possibly with some modification such as encryption, presenting an outer header structure that may differ from that found internally.

encapsulated packet characteristics. Systems such as packet filters would therefore have to make decisions based on the visible headers, or would have to look deeper into the packet − a process made difficult or infeasible by fragmentation, processing overheads and encryption.

Effectively, tunnelled data is currently impossible to reliably firewall, and may have to be considered only as trustworthy as the packet source.

2.1.3. IP Options

IP options form a way for implementations to communicate data in addition to that defined in the standard IP header fields. From a security point of view, the most significant of these are the Source Routing options.

Source Routing comes in two varieties: Loose (where arbitrary intermediate nodes may be visited) or Strict (where only the listed nodes may be visited). Originally intended for network debugging purposes, these have found use in supporting spoofing attacks.

In a conventional spoofing (where the source of the packet differs from that shown in the IP header), the attacker must work blindly (since replies will be returned to the spoofed address), or be on the routing path of a packet. The use of source routing allows attackers to place themselves on the packet's routing path, allowing them to effectively spoof external addresses. For this reason, blocking out packets with source routing, or simply removing those options at the firewall, may be advisable.

2.1.4. IP Addressing

Every device capable of transmitting onto an IP network must have a unique IP address. This address takes the form of a 32−bit word, generally represented as four decimal values (one per byte) separated by dots (the "dotted−decimal" representation).

In addition, IP devices are grouped into networks, which all share a common address prefix and can directly communicate without the need for an intermediary gateway. This network is represented as the number of bits in the common prefix, written to follow the device IP address (separated by a "/"). Alternatively, this can be represented by a dotted decimal address with 1's set in bits valid in the prefix, also known as the network mask.

For example, this document was written on a machine known as 132.181.8.5/21 or

132.181.8.5/255.255.248.0. Performing a logical AND on the latter representation's two dotted decimal values results in a network address of 132.181.8.0 − which is also the first 21 bits of the host IP address.

Any device with a 132.181.8.0 address prefix is directly reachable from this host. Any other hosts can only be reached via a gateway − in this case 132.181.8.254. That gateway is connected to multiple networks (each connection having an IP address and network prefix), and will pass packets on to another gateway (and so on), or on to the applicable host. This is also the mechanism used to allow firewalls to inspect all traffic − all packets between an internal address and an external address must pass through the firewall gateway.

Historically, the IP address range has been divided into classes ranging from A to E (Table 4).

Class	***Binary Prefix***	***Byte 1 Range***	***Implicit Network Mask***	***Significance***
A	0	0−126	/8	
B	10	128−191	/16	
C	110	192−223	/24	
D	1110	224−239		Multicast
E	1111	240−255		Reserved

Table 4. IP Address classes

This scheme has mostly been superseded by the introduction of Classless Inter−Domain Routing (CIDR) [CIDR] and subnetting [Subnet] as described above, and is now notable only for classes D − the multicast address range − and reserved class E. A number of such address special cases occur in IP − refer to Table 5.

Network	***Host***	***Relevance***
	255.255.255.255	Local network broadcast (dropped by routers)
127.0.0.0/8		Internal loopback address
	0.0.0.0	Source IP address unknown (bootstrapping)
0	*HostID*	Specific host on this network (bootstrapping)
NetID	*All 1's*	Directed broadcast to all hosts on N*etID*
NetID	*All 0's*	Directed broadcast to all hosts on N*etID* (outdated)
10.0.0.0/8		Private address space[6]
172.16.0.0/12		Private address space

6 Private address spaces were introduced in [PrivAddr] as a mechanism for allowing internal networks to use IP numbers without risk of addressing conflicts. Packets with such addresses are dropped by backbone routers.

Network	*Host*	*Relevance*
192.168.0.0/16		Private address space
224−239.0.0.0/8		Reserved multicast address space (class D)
240−247.0.0.0/8		Reserved address space (class E)

Table 5. IP Addressing Special Cases

Given the scale of the modern Internet, the available network address space is quite limited − especially when one considers the fact that address allocation blocks are often sparsely populated (at the highest level, address blocks were allocated according to the class structure described above). A solution to this problem is Network Address Translation.

2.1.5. Network Address Translation (RFC1631)

Network Address Translation (NAT) [NAT] is a technique whereby a gateway rewrites packet addresses to correspond to an illusionary network configuration. In other words, it rewrites packets passing through a gateway from an internal view to an external one, and vice versa.

Network address translation falls into two classes:

- **Address pool NAT** (a.k.a. Basic NAT), is where internal addresses are mapped to a pool of external addresses. The mapping consists of associating an internal IP address with an external one − all connections from that internal address are mapped to the same external address for the duration of the session.
- **Port pool NAT** (a.k.a. Network Address Port Translation, NAPT), is where individual connections are mapped from an internal (IP, port) pair to an external (IP, port) pair − where the external IP is generally that of the gateway itself.

NAT can also be classified by the direction of translation:

- **Static NAT** modifies the destination address to a fixed other address. This is most often used to translate incoming requests to a server's private IP address. Port pool NAT remaps requests to a given port on the gateway, while address pool NAT remaps requests to a given external address. Packets in the inverse direction are similarly remapped using the inverse mapping.
- **Dynamic NAT** modifies the source address of packets. For port pool NAT, the new source is a random port on the gateway (which maintains a record of the mapping to allow responses), while address pool NAT changes the source IP of packets to match a free external IP. Packets in the reverse direction just use the

inverse mapping.

Consider, for example, packets between 10.0.0.1 (an internal host) and 64.0.0.1 (an external host). The NAT device has an external IP address of 200.0.0.254/24, and may use any other addresses on that network for address pool mapping. 200.0.0.80 may also be published as a web server IP address. Table 6 shows possible packet mappings for the different classes and directions of NAT.

Original Request	***NAT Pool Class***	***NAT direction***	***Mapped Request***
10.0.0.1:1025 −> 64.0.0.1:80	Address	Dynamic	200.0.0.37:1025 −> 64.0.0.1:80
10.0.0.1:1025 −> 64.0.0.1:80	Port	Dynamic	200.0.0.254:49953 −> 64.0.0.1:80
64.0.0.1:4093 −> 200.0.0.80:80	Address	Static	64.0.0.1:4093 −> 10.0.0.1:80
64.0.0.1:4093 −> 200.0.0.254:80	Port	Static	64.0.0.1:4093 −> 10.0.0.1:80

Table 6. NAT translation examples

Static NAT is a mechanism for allowing access to private servers from external hosts, or to allow private address ranges to be used with Internet connectivity. For example, this could be used to avoid having to change internal IP configurations after changing ISP. Dynamic NAT is used to distribute clients across IP addresses − typically serving an entire network using only the gateway's external IP address. Dynamic NAT can also be used for load−balancing − see Section 9.3.1.

All forms of NAT (except the simplest forms of static NAT) require the gateway to keep track of which mappings are in use at any point − essentially requiring the gateway to maintain connection states for all connections (or simulate this for connectionless protocols).

Most importantly, NAT allows organisations to use the private IP address ranges listed in Table 5 while maintaining Internet access. A small number of public addresses can be used to support a large number of private devices.

2.1.6. Address Resolution Protocol (RFC826)

The Address Resolution Protocol (ARP) [ARP] is used by network devices to translate network layer IP addresses to data link layer Ethernet addresses.[7] When a device attempts to communicate with another device on the local network for which it does not have a link layer (MAC) address, it first broadcasts an ARP request for that address − listing its own MAC and IP addresses, and the IP address sought. On receiving an ARP request for its own IP address (or one that it will pass data to, in the case of Proxy ARP), a device responds with its own MAC address.

A device receiving an ARP response will update its internal address tables unless a conflicting entry is statically set − no matching request is required (as is the case with gratuitous ARP responses).

2.2. User Datagram Protocol (RFC768)

The User Datagram Protocol (UDP) [UDP] is essentially a transport layer wrapper to the IP service. The only features offered on top of those provided by IP are:

- Process−to−process addressing, using UDP ports. Every host has 65535 UDP ports per IP address, each potentially identifying an application.
- Optionally, data integrity checking in the form of a "16−bit one's complement of the one's complement sum" of a pseudo IP header (see Table 7), and the complete UDP packet (0−padded if necessary). This can be set to zero if no checksum is required.

As such, UDP can be described as a connectionless, unreliable application−to−application transport protocol. It is used in applications that do not require, or cannot afford, a reliable connection setup, or that need to broadcast or multicast packets. Such applications must internally handle the cases of packet loss, out−of−order delivery and packet duplication.

UDP packets take the form of a data segment prefixed by the UDP header (shown in Table 8).

	0		*2*		*4*		*6*		*8*		*10*		*12*		*14*		*16*		*18*		*20*		*22*		*24*		*26*		*28*		*30*	
0	Source IP Address																															
4	Destination IP Address																															
8	Zero														Protocol								Packet length									

Table 7. IP pseudoheader used in TCP/UDP checksum calculations

7 Typically, though ARP has a more general application in principle.

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	Source Port								Destination Port							
4	Length								Checksum							
	Data															

Table 8. UDP datagram header format

Since UDP does not have an explicit concept of connection, supporting UDP protocols in systems that rely on connections, such as NAT, is difficult. Possible solutions include application−specific tracking modules (as used for complex TCP applications), or some type of implicit connection. In the latter case, a UDP packet with similar addressing to a previously seen packet (e.g. a reply to a recent request) is assumed to be part of the same conversation and handled accordingly. This association persists until some timeout expires (or until some series of packets are seen).

Complications with this approach include handling long−lived, low bandwidth connections, and handling multicast or group communication. Application−specific tracking modules solve these problems, but need to be developed for every distinct application protocol.

2.3. Transport Control Protocol (RFC793)

The Transport Control Protocol (TCP) [TCP] extends the services provided by IP to include application−to−application addressing, and a reliable, ordered byte stream connection. This basis allows many of the complex protocols in use in modern networks.

As with UDP and IP, TCP encapsulates user data segments by prefixing the header shown in Table 9.

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	Source Port								Destination Port							
4	Sequence Number															
8	Acknowledgement Number															
12	Data Offset		Reserved			Flags			Window							
						URG	ACK	PSH	RST	SYN	FIN					
16	Checksum								Urgent Pointer							
20..	Options (if present) + Padding															
	Data															

Table 9. TCP segment header format

The fields in the TCP segment header are:

- *Source Port* − Identifies the application from which data was sent. Though similar to UDP ports, TCP ports fall into a disjunct address space − different applications may be using the same port number for UDP and TCP. A TCP connection is fully identified by the (source IP, source port, destination IP, destination port) tuple − multiple instances of the same application may receive data on the same port from different remote endpoints, and an application can register to receive connections on any one, or all, of the IP addresses associated with a host.
- *Destination Port* − Identifies the application to which data is being sent.
- *Sequence number* (SEQ) − Identifies the position of the first data byte in this segment, relative to the initial SYN packet.
- *Acknowledgement Number* (PACK) − Identifies the next sequence number expected by the sender. This piggyback acknowledgement implies that all previous sequence numbers have been correctly received.
- *Data Offset* − The number of 32−bit words in the header − typically five, but possibly more if TCP options are present.
- *Reserved* − Reserved for future use: must be set to zero.
- *Flags* − Contains connection control and packet delivery modifiers:
  - *URG* : When set, the *Urgent Pointer* field is valid.
  - *ACK* : When set, the *Acknowledgement Number* field is valid.
  - *PSH* : When set, the data in this packet should be delivered immediately, avoiding buffering in the destination TCP handler.
  - *RST* : When set, the receiving socket is reset. Effectively forces a desynchronised connection to close.
  - *SYN* : When set, this flag indicates that the sequence number in this packet represents the initial (base) sequence number for this connection. The first data byte will be SEQ+1 (the SYN flag consumes one sequence number). Only valid during connection setup.
  - *FIN* : When set, this flag indicates that the sender has no further data to transmit beyond this segment. Like a SYN packet, a FIN consumes one sequence number, and must be acknowledged.
- *Window* − Indicates the number of bytes that the sender is willing to accept, starting at PACK. Used in flow control to limit unacknowledged data.

- *Checksum* − The "16−bit one's complement of the one's complement sum " of a pseudo IP header (see Table 7), and the complete TCP packet (0−padded if necessary).
- *Urgent Pointer* − TCP offers an "out−of−band" service for delivering priority data. If the URG flag is set, this pointer indicates the offset of the first byte of priority data, which should be delivered to the application asynchronously. In practice, delivery of urgent data is poorly supported in the sockets interface [Stevens−1], and depends on applications. This feature has been used in an attack on software incapable of handling priority data. [WinNuke]
- *TCP Options* − Contains extension information to the general TCP header. To ensure backward compatibility, extension TCP options may only be used if present in both SYN packets that set up this connection. Some of the TCP options in use are:
  - *Maximum Segment Size* : Sent with a SYN packet, this specifies the maximum segment size the sender is willing to accept.
  - *Window Scale* : Specifies the number of bits by which the TCP Window field should be scaled up in order to support high−speed and high−latency connections.
  - *Timestamp* : Contains packet timestamps used to avoid sequence number wrapping ambiguities. However, due to the PAWS[8] [TCP−HP] mechanism in TCP, if the value of this option differs from that expected, it may lead to a packet being discarded. The *prism* TCP hijacking module used in this thesis therefore strips out this, and any other unrecognised TCP options.

  TCP options take the form of a single option−kind byte, or an option−kind byte followed by an option−length byte, followed by option data. The use of TCP options presents a problem to connection resumption, since retrieving the correct options (potentially including unknown extension options) in use on a connection is impossible. While this may benefit from further investigation, in general we have chosen to remove unknown or problematic options when recreating connections.
- *Data* − The segment payload, identified by the Destination port field.

8 Protection Against Wrapped Sequences, supporting connections with a large bandwidth and high packet latency. Essentially, TCP sequence numbers are extended by the value of the timestamp option.

2.3.1. TCP Connection Flow Control

TCP includes a variety of flow control algorithms, including mechanisms for ensuring reliable delivery, segment ordering, congestion control and connection termination. For a fuller description of these mechanisms, refer to the applicable RFCs or a good book on networking (such as [Stevens−1] or [Stallings]).

In particular, TCP guarantees [Zwicky]:

- Ordered delivery,
- Complete delivery,
- That no duplicates will be delivered to the application.

Maintaining these guarantees in spite of node failures is difficult, and is further discussed in Section 7.

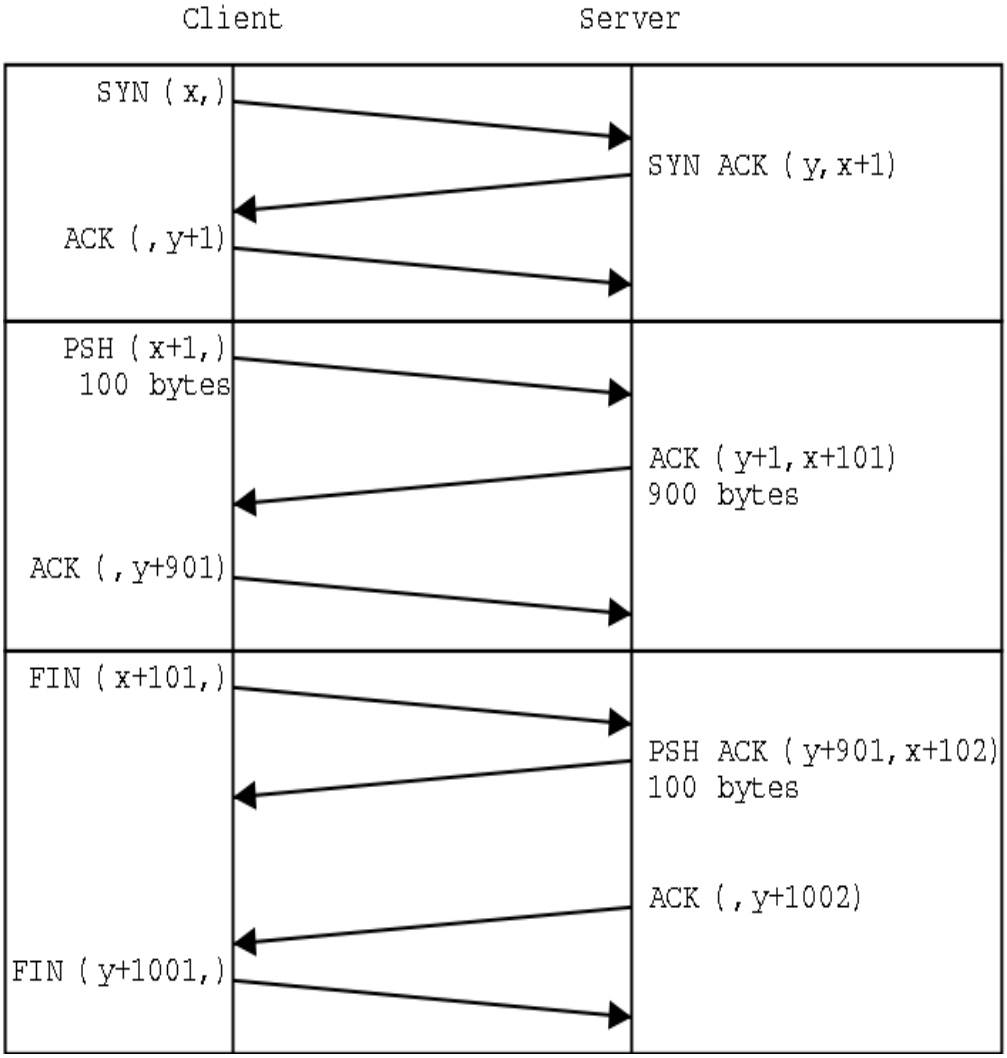

Figure 3. TCP Connection setup, simplified data transfer, and connection termination

As a brief summary, Figure 3 outlines a simple request/reply scenario: the client

connects to the server, transfers a 100−byte request and closes down its end. The server receives the request, sends 900 bytes (which the client acknowledges), receives the client close, completes its reply (piggybacking an acknowledgement of the client close), and completes the connection close.

In the context of this thesis, important aspects of this process are:

- The sequence and acknowledgement fields of a packet must match that expected by the destination. If not, the receiving endpoint responds with an ACK packet containing its current correct acknowledgement number. This synchronisation between endpoints offers TCP some protection against data loss or injection, as described in Section 3.2.2.
- The acknowledgement number sent by an endpoint represents the first data byte in the connection stream guaranteed not to have been delivered to an application. This represents the starting point for recovering that connection stream.

2.4. Internet Control Message Protocol (RFC792)

The Internet Control Message Protocol (ICMP) [ICMP] has a dual function: it serves as a carrier for debugging information and applications (such as *ping* and *traceroute*), and carries error messages generated by the use of other IP−based protocols. As such, it forms a fundamental part of the functioning of the Internet, and in particular of TCP.

Some of the common ICMP message types, with short descriptions, are listed in Table 10. Table 11 outlines the general format of ICMP packets − though different ICMP types often have distinct formats.

ICMP Type	***ICMP Code***	***Name***	***Significance***
3		Destination Unreachable	
	0	Net Unreachable	A packet was discarded due to a routing problem (no route to destination known), an attempt to access an inactive port or protocol, or an inability to deliver the packet while complying with the packet header. Blocking these messages would break IP−based protocols. ICMP Code 4 messages are also used in TCP's path MTU discovery mechanism.
	1	Host Unreachable	
	2	Protocol Unreachable	
	3	Port Unreachable	
	4	Fragmentation needed and DF set	
	5	Source Route failed	
11		Time Exceeded	

ICMP Type	*ICMP Code*	*Name*	*Significance*
	0	TTL exceeded in transit	A packet was discarded due to its TTL field dropping to 0 (initial TTL set too low or routing loop), or a fragmented packet could not be reassembled in time − probably a fragment loss.
	1	Fragment reassembly time exceeded	
12		Parameter Problem	
	0	The packet could not be processed due to a problem in the packet headers, and was discarded. A pointer in the ICMP header indicates where in the packet (an extract of which is included in the message) the problem occurred.	
4		Source Quench	
	0	A host along the delivery path is discarding packets due to overloading, or in danger of doing so. The sending application should reduce transmission rates − though some implementations ignore this. The associated packet may or may not have been discarded.	
5		Redirect	
	0	Network Redirect	A redirection message is sent when the next hop and packet source for a message are on the same network − shortening the transfer path. If the packet has a source routing option present, no redirect is sent.
	1	Host Redirect	
	2	ToS and Network Redirect	
	3	ToS and Host Redirect	
8	0	Echo	On receiving an Echo message, a host replies with an Echo Reply, using the same sequence number and identifier. Used by *ping* to test connectivity, but also used for network mapping.
0	0	Echo Reply	

Table 10. Some ICMP message types and descriptions

	0		*2*		*4*		*6*		*8*		*10*		*12*		*14*		*16*		*18*		*20*		*22*		*24*		*26*		*28*		*30*	
0	Type								Code								Checksum																	
4	ICMP Type Parameters																																	
8 ...	ICMP Data: usually the IP header + 64 bytes of the problem packet																																	

Table 11. ICMP General packet header format

Other ICMP messages include the timestamp request and reply, the information request and reply messages (used to identify the local network − the request may source and destination address as 0, and the reply will have full addresses), and the address mask request and reply messages [Subnet].

ICMP messages can be divided into informative messages (echo, timestamp request, etc), and error messages (destination unreachable, time exceeded, etc). Informative messages are often blocked by firewalls, reducing an attacker's ability for network mapping, but error messages should not be blocked. ICMP error messages are subject to abuse, however. One technique used in connection hijacking is to send spurious

ICMP redirect messages to hosts, advertising the attacker's address as a shorter route to the local gateway [Redir]. The attacker can then forward and modify packets on that route at will.[9]

Many of the ICMP error messages (notably types 3, 11, 12, 4, and 5) include the IP header and first 64 data bytes (which typically covers the transport protocol header as well) of the problem packet. In load−balancing and packet filtering systems (refer to Sections 5.2 and 9.3.1), these ICMP packets should be regarded as part of the same connection as the failed packet in order for IP−based protocols to function correctly.

2.5. Application Protocols

A number of application protocols are listed in Figure 2. In this section, we will briefly discuss interesting features of some of these protocols. For a more complete discussion please refer to networking references such as [Stallings] or [Stevens−2].

2.5.1. Single−Connection Protocols

A common theme among TCP−based services is that a client opens a TCP connection to the server, and all further communication occurs across this single connection. Examples of protocols that use this model include HTTP, Telnet, SMTP, rlogin (and derived services) and the Unix diagnostic services (echo, discard, etc.).

The client port is typically in the unprivileged range[10] (rlogin is a notable exception, where the client port must be in the privileged range 512−1023), whereas the server port is generally the IANA assigned port number for that service. This behaviour makes these protocols simple to packet filter and handle in load−balancing systems.

2.5.2. Multi−Connection Protocols

Some protocols use multiple TCP connections:

- FTP [FTP] is an example of a protocol that uses a predictable control connection (client port −> port 21), and which signals variable data connections to the client using that control port. Data connections can then follow as either:
  - Passive mode FTP, where the client opens another connection (client port −> port 20) to the server, or

9 However, since ICMP Redirects are only valid within the source network, they can be safely filtered.

10 Ports 1−1023 are termed privileged, since they are only available to software running with administrative privileges. With the proliferation of single−user machines, however, this distinction no longer offers any security.

- Active mode FTP, where the client nominates an address[11] which the FTP server connects to. (server port −> client port)

- The RPC *portmapper* service resides on a fixed port, and is contacted using single−connection semantics. The client uses this server to identify the port numbers used by other RPC−based services (such as NFS) on that server − which are then contacted separately. From a firewalling, NAT and load−balancing point of view, these connections are related, and must be handled accordingly.

These services present a problem to connection−aware gateway services, since the information used to associate connections can only be extracted from the initial connection. For NAT systems that change the address or port of connections, the problem is worse − any addressing information sent across the control connection must be rewritten to match the lower−level NAT addressing changes.[12]

DNS [DNS] is an interesting case among multi−connection protocols. Generally, a DNS request and reply takes the form of single UDP packets (in order to optimize performance). Longer DNS responses, and DNS zone transfers, are carried across persistent TCP connections. This raises two issues:

1. In order for DNS queries to be load balanced and proxied, both the UDP conceptual connection and the TCP connection should be channelled through the same handler node.
2. DNS is an example of a service that can maintain TCP connections for an indefinite period, despite having no traffic flowing. Connection−tracking schemes that use connection timeouts (NAT, stateful packet filtering, IDS) must take this behaviour into consideration.[13]

Another variation on the theme is multicasting protocols. A common thread among the previous examples, used as a heuristic by some systems, is that the IP addresses of the communicating peers are fixed. In a multicasting conversation, this no longer holds − replies associated with the same conversation could come from any remote address. The destination address should remain the same, however − though that will fluctuate between being an internal address (for incoming messages) to being an

11 Generally some unprivileged port on the client. By nominating a third party address (such as a vulnerable service reachable from the FTP server), the FTP server could be used to launch attacks. [FTPBounce]

12 This is a difficult issue to solve efficiently. The Linux Netfilter system includes a good example of how this is done in practice, and illustrates some of the complexities involved [Netfilter].

13 [Requirements] notes that TCP Keepalive messages are optional, and if implemented must default to wait at least two hours before taking effect. TCP connections do not automatically time out.

external address (for outgoing messages) with respect to routing.

3. Network−based Attack Techniques

3.1. Sources of Attack

Script Kiddies: This is the name used for relatively unskilled hackers that use others' pre−written tools. They are typified by having endless time to spend searching networks for victims to their latest exploit[14] − it is on these that the common perception of hackers is based. [GRCDoS] contains an excellent example.

This is not to say that they do not pose a risk, however − far from it. These hackers often have an array of tools available, and keep up to date with the latest new exploit software that becomes available. In addition, since they often have no specific aims in mind (beyond the trophy of having hacked a system), they will not necessarily target the most visible or valuable machines − obscurity is no defence.

Employees: Possibly the most dangerous group of potential attackers are the very people who use the networks every day − the staff. They know what in a network is of value, what defences are in place, and have a ready foothold from which to escalate their control. Though the figure has been dropping since 1998, 76% of respondents in the 2001 CSI/FBI survey [CSI] still consider disgruntled employees a likely source of attack, with 31% reporting attacks on internal systems.

Mistakes: Not all anomalies in networks have hostile intent. Many "attacks" might be result from a lack of user expertise or from simple user error. This does not imply that such errors are not dangerous: the case of the 1980 ARPAnet collapse [ARPAnet] is a clear example of how devastating a simple mistake can be.

Automated Agents: This category includes such things as worms (such as the infamous 1988 Internet Worm [Spafford], or the more recent Lion [Lion], Ramen [Ramen] and sadmind/IIS [IISWorm] worms[15]), viruses, and trojan software [BackOrifice][16].

Expert hackers: A number of expert hacker groups have been in the media over the past few years − as government witnesses, software developers, and network security experts. These groups do not merely use exploits written by others; they produce tools of their own[17]. They constitute the highest skill level that network security will be faced with; an administrator can expect to see completely new attacks, if any signs

14 Getting hold of such tools is surprisingly simple − for example, see ftp://technotronics.com.
15 See http://project.honeynet.org/papers/worm/ for a detailed analysis on worm behaviour.
16 See [Kyas] for the differences between worms, viruses and trojans.
17 For example, [Hobbit]

remain at all. [HTTPShack]

The objective of most of these attackers is to compromise the victim system. To this end, they will employ predominantly application−level attacks: DoS and network level attacks have their place, but offer less of a return in terms of this goal.

The reason behind a given attack may differ wildly: recreation, industrial espionage, fraud, and attempts by foreign governments to destabilise national infrastructure[18] have all been proposed as causes for intrusions. [Joyal] [Kyas Chapter 2] [Law&Net] [Denning]

To place this discussion into context, consider some specific reports:

- "However, the hackers of the cases on which this paper is based are known. All of them were male, and computer science students doing their master's. They all had access to the Internet, and were reasonable well acquainted with UNIX. All of the hackers, except one, had the level of an ordinary UNIX programmer with a little bit more understanding of network software" [Doorn]
- "A sixteen year−old from the U.K. entered a plea bargain and paid a $1900 fine while another twenty−two year old pled not guilty and was acquitted on all charges in February 1998. The 16 year old was operating on a home computer in his parents' house and had a "C" grade average in his high−school computer class" − Rome Labs, March 1994 [DOD]
- "The attackers were two teenagers from California and one teenager from Israel. Their motivations were ego, power, and the challenge of hacking into U.S. DoD computer systems." − SOLAR SUNRISE, February 1998 [DOD]
- "hi, its me, wicked, im the one nailing the server with udp and icmp packets, nice sisco router, btw im 13, its a new addition, nothin tracert cant handle, and ur on a t3.....so up ur connection foo, we will just keep comin at you, u cant stop us "script kiddies" because we are better than you, plain and simple." − grc.com, May 2001 [GRCDoS].[19]

It would appear as if the common preconception of hackers being young, male and bored holds. However, real information is scarce − though a question would be whether experienced hackers are caught. What is clear, however, is that attackers use increasingly complex tools and are often completely indiscriminate in their target

18 Consider the recent series of hacking activity between US and China [Hackwar].
19 Based solely on these reports, the age of the attackers show an interesting pattern.

selection.

3.2. Forms of Attack

3.2.1. Passive and indirect probing

The first step in attacking a system includes attempting to gain as much information about the prospective target as possible. This may range from simple reachability and service checks, to full−blown background research on an organisation. Active probing is covered in a later section, but a variety of information sources are accessible using indirect or benign (and therefore virtually undetectable) techniques [Bartley]:

- **Whois** [Whois]: The *Whois* database contains a record of IP address allocations, including the official name, address information, administrator contact information and name servers. Versions of this database are maintained by Network Information Centres (NICs), and can be useful in backtracking attackers, or may expose sensitive configuration details.
- **DNS**: The domain name service contains name to address mappings for Internet hosts. This includes IP address(es), primary and secondary host names, and possibly administrative contact details. This information can be used to identify and classify potential targets.
- **Service banners**: Many servers offer information about the software and version they run, sometimes including (implicit or explicit) information on the host operating system. This information is readily accessible when using services − loading a web page from the site, bouncing an email message from a server, etc. − which may be indistinguishable from normal usage.
- **Passive OS signatures**: A technique developed for analysing hacking attacks [PassiveID], this allows identification of the operating system (often including the specific version number) involved in a network communication, based on captured traces. Every network stack implementation has unique features − by identifying these features in network traces, a guess can be made as to the implementation involved[20]. This allows attackers to classify the remote operating system in use, possibly identifying security devices, by comparing this information to that derived from service banners.

Other techniques include using *traceroute* to identify service providers, public

20 This is similar to the operating system signature scan technique used in *nmap* [nmap], discussed in Section 3.2.2.

information on organisational websites, and predictable host and user names.

3.2.2. Network Level Attacks

3.2.2.1. Active probing

Active probing of a network involves sending network traffic to potential target locations (IP address and service ports). Responses are compared with expected behaviours to deduce the presence of potential targets. Some specific techniques, many of which are implemented in tools such as *nmap* [nmap], include:

- **TCP connect() scan**: the scanning device attempts to open a TCP connection to each scanned address/port combination, testing for accessible services.
- **TCP SYN scan**: the scanning device sends TCP packets with the SYN flag set. Available services respond with a SYN/ACK. This technique may bypass application−level access controls, since the TCP handshake is never completed, but may be picked up as a SYN flood.
- **TCP ACK scan**: the scanning device sends TCP packets with the ACK flag set, simulating an acknowledgement of an existing connection. This technique may penetrate naive packet filters, and tests for host existence and active (though inaccessible) services.
- **TCP NULL, Xmas, FIN scans**: the scanner sends TCP packets with (respectively) no flags, FIN + PSH + URG flags, and FIN flags. These packets attempt to bypass more sensitive packet filters.
- **UDP scan**: the scanner sends 0−length UDP packets; closed ports return an ICMP error message. Since UDP is stateless (and less common in attacks), some firewalls fail to filter UDP correctly.
- **ICMP Echo scan**: the ICMP echo message is a simple method to test connectivity, but is filtered by many firewalls.
- **Fragmented scan**: used in conjunction with other single−packet scans, this technique uses IP fragments instead of contiguous packets. This may elicit responses through firewalls with naive handling of IP fragments.

  Another variation on the fragmented scan lies in combining this with the overlapping fragments problem described in Section 2.1.1. Filtering devices with permissive assumptions about outgoing traffic may be bypassed using this technique.

- **Operating System identification** [OSID]: Each network stack implementation has unique features. By sending a variety of valid and invalid packets, a scanner can deduce which implementation is present on an address, often including operating system and version. Because of the use of incomplete connections and invalid packets, this scanning technique is more accurate and much more easily detected than the passive technique described earlier.

3.2.2.2. Denial of Service

Denial of service (DoS) attacks can be classified under two main varieties: technical attacks, where a service is disrupted by an incomprehensible request, or flooding attacks, where a service is disrupted by a massive surge of requests. Flooding DoS attacks often require support from technical aspects − the distributed DoS (DDoS) attacks during February 2000 [DDoS] included distributed clients and use of the Smurf attack.

While techniques have been developed to mitigate the effects of DDoS[21] attacks [DdosFaq][22], no practical complete solution is available. We will therefore focus on technical DoS attacks.

- **Smurf** [Smurf]: an attacker sends a flood of ICMP echo request packets to an unfiltered directed broadcast address, with a source address as that of the attack victim. When all hosts on the broadcast network respond, numerous echo response packets (for every echo request sent) are sent to the victim, overloading its network link.[23]
- **Land** [Land]: an attacker sends a SYN TCP packet to the victim host with identical source and destination addresses. Unless correctly handled, the looping responses sent lock up the victim's operating system.
- **Nestea** [0−frag], **Ping of Death** [PingOfDeath]: an attacker sends a series of illegally fragmented packets to the victim. Attempting to reassemble these leads to buffer overruns inside the IP handler, crashing a vulnerable system.
- **SYN flood** [Schuba][SYNflood]: an attacker sends a large number of SYN packets to a TCP service. Since a server can queue a limited number of incomplete

21 http://grc.com/dos/grcdos.htm contains a fascinating description of a DDoS attack on grc.com during May 2001. In this case the attack could be filtered out, since it used real IP addresses and predictable traffic. As attackers and host network implementations become more complex, this is not always possible.

22 Ranging from detecting and blocking these attacks at network ingress points, to changing network connections and addressing to avoid the flood.

23 http://netscan.org/ contains a list of such unfiltered networks, also known as smurf amplifiers.

connections, and each connection must be held for some length of time, this attack can prevent a server from responding to real requests.

- **WinNuke** [WinNuke]: an attacker sends TCP urgent data to a NetBIOS service. Old versions of Microsoft Windows are incapable of handling this data and crash on receipt.

3.2.2.3. Address spoofing

Address spoofing is the practice of producing IP packets where the source address does not match that of the interface that produced it. This technique has many general applications, including transparent proxies and (arguably) NAT. It can also be used to obscure the source of simple attacks, and to bypass address−based access controls.

Address spoofing is, in general, only useable for unidirectional communication, unless the packet source also controls the network route between the destination and source (as is the case in transparent proxies on firewalls)[24]. By default, responses to spoofed packets are sent to the apparent source.

Eliminating spoofing of internal addresses is possible by the application of filtering rules on network gateways. Identifying spoofed external addresses is more difficult[25], infeasible in some cases. Implementation of tunnelling protocols may facilitate address spoofing (as is used for PPTP), but has been identified as a possible attack mechanism [FW−1Eval].

Another form of spoofing lies in the disjunction between addressing information used on the network and application layers. A simple example is SMTP message spoofing: using a connection from an external host and injecting a message purporting to be from an internal source. Similarly, some HTTP proxies will serve external hosts requesting remote content (for example, using an external proxy to retrieve overseas content), allowing these hosts to avoid local usage logging and charging.

Address spoofing is generally associated with DoS attacks, and TCP session hijacking (Section 3.2.2). Transparent proxies (Section 5.2.5) also use a controlled form of address spoofing, as does our *prism* connection recovery module (Section 8).

24 A much more advanced technique, blind spoofing, requires an attacker to be able to predict responses, effectively simulating response packets [TCPHijack].

25 One possible approach would be to compare incoming packet signatures to that expected for the apparent source. This would require significant tracking, however, and is probably only useful in combating TCP connection hijacking.

3.2.2.4. TCP session hijack

The concept of TCP session hijacking is not new [Morris]. It is comprised of techniques used to inject data into TCP connections, effectively taking control of that connection. Since most (non−cryptographic) TCP−based protocols rely on authentication at connection setup, and continue to apply the privileges gained at that point for the life of the connection, this allows an attacker to obtain any privileges associated with a connection.

TCP maintains reliable delivery using synchronised sequence number tracking on both endpoints, with a windowing acknowledgement scheme. This scheme also offers some protection against data insertion. Should the endpoints become desynchronised, however, there is no recovery mechanism.

As described in [TCPHijack], an attacker can use this feature to gain and maintain control of a TCP connection. The ability to do this depends on the ability to inject a TCP packet with appropriate sequence numbers into one endpoint − requiring the ability to predict or observe sequence numbers.

The ability to predict sequence numbers depends on the choice of initial sequence number, since later sequence numbers monotonically increase from that value. The randomness of the initial sequence numbers varies between TCP implementations − prediction difficulty ranging from trivial − an implementation using fixed or periodic numbers − to infeasible[26]. A firewall that rewrites TCP sequence numbers (such as a proxy firewall) will increase difficulty to the level of its own TCP implementation.

The second option, being able to observe sequence numbers, relies on the attacker having access to the packet routing path. Hunt [Hunt] is a software tool that implements session hijacking using ARP spoofing and ICMP redirects to modify packet routing to pass through an attacking host[27]. Currently, the only effective defence against this form of attack is in the use of cryptographic protocols, or securing the entire routing path (including source and destination networks).

In our implementation of a TCP session recovery system (*prism*), we make use of this capability to effectively hijack disrupted TCP sessions. The implementation of a general TCP hijack framework (based on our *prism* component) was suprisingly simple. Refer to Section 8 and Appendix 3 for more detail.

---

26 [TCPISN] outlines the concerns of predictable TCP initial sequence numbers in the light of recent statistical evaluations.

27 See [MiMattack] and [TelnetHijack] for detailed traces and analysis of this form of attack.

3.2.3. Application Level Attacks

With the possible exception of TCP hijacking, none of the network level attacks described in the previous section are capable of leading directly to a server compromise[28]. In addition, many of these attacks can be effectively detected and blocked at the firewall − in particular, a proxy firewall will block most of these attacks automatically. Similarly, a good firewall (in particular proxy firewalls) will limit the effectiveness of active scanning techniques to the firewall and public services.

Application level attacks are far more of a problem. These attacks use weaknesses in complex application software, and may lead to direct server compromise. Two categories of direct application level attack can be distinguished:

3.2.3.1. Application buffer overflows

Most application protocols include remotely generated data fields at some point: HTTP request URLs, FTP file and directory names, DNS request addresses and server responses. Often, this data is extracted from the network stream, stored locally, and processed from there.

When storing data values, there is often an implicit capacity − the size of the internal server buffer for that field. If the data length exceeds that buffer size, and the software does not truncate the data during storage, memory areas directly following the buffer will be overwritten. What was stored in that memory area depends on the exact server implementation: it may include other data fields (possibly invalidating previous field validations), function return locations, or executable code. [Aleph1]

A buffer overflow can have a number of results: it may have no visible effect, it may crash the server (acting as a DoS attack), or it may allow an attacker to execute arbitrary code using specially crafted data.

A filtering firewall has no simple method of preventing this form of attack. A firewall proxy, through its understanding of the protocol's field semantics, can sanitise protocol fields (blocking requests or trimming excess data). This requires more complex proxy implementations, and walking the line between blocking attacks and allowing server functionality may be difficult.

[Forensic] describes a buffer overflow on a *bind* DNS server. An attacker requests a

28 In other words, is capable of allowing an attacker to execute arbitrary commands on the remote server.

recursive DNS lookup for an external address he controls from the victim DNS server. When the victim contacts the attacker's DNS server to resolve the query, the attacker's modified server replies with a TCP connection carrying a large block of data − which overflows the DNS server application, crashing it and opening an interactive session. The attacker can now log into the DNS server with the same privileges that the DNS process had − typically administrative privileges.

3.2.3.2. Application design and implementation failures

The second class of application attack uses legal but unexpected input to servers. What happens based on this input varies greatly − at worst, it allows an attacker the ability to execute arbitrary system commands as the privileged user.

Some examples of such attacks include:

- **WinNuke** [WinNuke] : When a vulnerable NetBIOS implementation receives TCP urgent data, it crashes the system. TCP urgent data is a common part of the TCP protocol, used to support applications such as Telnet, but is inappropriate when directed at a NetBIOS server.
- **URLs containing device files** [Securax] : By passing a URL containing references to special devices files (eg. C:\CON\CON) to vulnerable MS Windows 95 and 98−based web browsers, an attacker can cause that host to crash.[29]
- **Back doors** : A back door is a special access mechanism implaced by system designers to bypass normal security controls, as described in [BackDoor]. Once such a back door is discovered, any attacker can use it to bypass security.
- **Parameters containing special characters** : Many applications (intentionally or not) interpret some characters found in parameter values specially. A simple example is '../', when embedded into HTTP URLs, which may allow an attacker to retrieve files outside the normal web page structure. Refer to [WWWSecFaq] for more examples.

These attacks may be impossible to block in general. Firewalling proxies with detailed information on the exact capabilities and weaknesses of the service and server in question may be able to sanitise input presented to servers. Some aspects of this goal can be handled via constraining input to known safe values, for example stripping all active content (Java, ActiveX and HTTP−based scripting) out of HTTP

29 http://www.cosc.canterbury.ac.nz/~mpj17/crash_windows.html can be used to test for this vulnerability.

traffic. This may lead to issues with supporting extended protocol functionality, however.

Another approach is to use a system similar to misuse detection in IDS [ANS] to recognise known attacks. These attacks can then be filtered out at the firewall, reducing the number of systems that must be hardened. This method, however, is computationally prohibitive for conventional firewall systems.

4. The design of a clustered firewall

In modern networks, firewalls often form the first, and sometimes only, line of defence against attack. However, it is in the nature of a firewall to act as a network bottleneck − all traffic to be controlled must pass through the firewall.

Modern networks are also getting faster − on all levels, the bandwidth available has been increasing. In particular, with the advent of cable and xDSL modems, private users have also gained access to high speed, stable connections. Finally, of course, the network population is still growing near−exponentially. As a result, the performance demanded from firewalls is increasing.

There have been three major responses to this trend. Firstly, the hardware running a firewall can be expanded − to the point of using large, multiprocessing systems [FWReview]. This approach clearly has limited scalability, and does not leverage pre−existing systems. Secondly, firewalls are increasingly using low−level filtering techniques (stateful packet filtering, connection tracking, NAT) in the place of slower, application−level proxies. As a third option, high−load systems can be placed behind separate firewalls (or not firewalled at all) − by partitioning networks, the load of individual firewalls can be reduced. This approach, however, often does not scale well and introduces a significant administrative overhead.

Another issue with modern firewalls is their inability to protect systems against application−level attacks on public services. Firewalls, while effective in protecting private systems and services, are generally incapable of recognising attacks using public services − such as HTTP−based attacks against an organisational web server. This problem is further exacerbated by the use of low−level filtering techniques, which do not allow fine−grained protocol control.

The obvious solution is to design more intelligent firewalls: firewalls that do more than use current filtering techniques. By merging the attack recognition capabilities present in Intrusion Detection Systems (IDS) into firewalls, it becomes possible to block attacks at all protocol levels.

Doing this would introduce two problems. First, firewall systems would become more closely bound to the protocols running across them, increasing the development costs. In many cases, however, stateful filtering techniques (refer to Section 5.2.3) are already highly protocol−specific. The second, more difficult, issue lies in the processing overhead associated with detailed protocol inspection. In order to

recognise protocol attacks, a firewall must closely inspect the traffic content − requiring the use of application proxies and extensive processing.

In this chapter, we propose a firewall design that would enable firewalls to scale to arbitrary processing power. A clustered firewall, while not necessarily increasing the aggregate bandwidth of a firewall, would allow sufficient processing power to be present in a firewall system to allow a firewall to fulfil its mandate: to protect internal systems against attack.

4.1. Requirements for a clustered firewall

In order for a clustered firewall to be useful, there are a number of requirements it must meet:

1. The system must be capable of basic firewalling functions. Typical modern firewalls support stateful and stateless packet filtering, NAT, transparent application proxies, remote administration and Virtual Private Networking (VPN). A firewall cluster should support all these functions as well.
2. The system should be structured so that it could be placed into the same network topology as a conventional firewall. In other words, the general network topology should not have to change to accommodate a firewall cluster. This includes support for proxy−ARP (drop−in) firewalls and demilitarised zones (DMZ).
3. The system must scale through a variety of system loadings, and should allow nodes to be added and removed with minimal, if any, reconfiguration.
4. The system must handle node failure elegantly; ideally, the system should maintain service in spite of node failures.
5. The system should utilise nodes efficiently. Current firewall fail−over schemes [CHA] [WGHA] make use of a backup node. The clustered firewall should attempt to ensure that nodes have a balanced load.
6. The system should not require client modification, and should ideally use existing firewall software and structures.
7. No one node or device should be a single point of failure.

4.2. Our design

The problem of developing a distributed, clustered firewall can be broken down into two parts: that of load−balancing, and that of fail−over control (Figure 4). While there is some interrelationship between these functions, they can be developed and

tested largely in isolation.

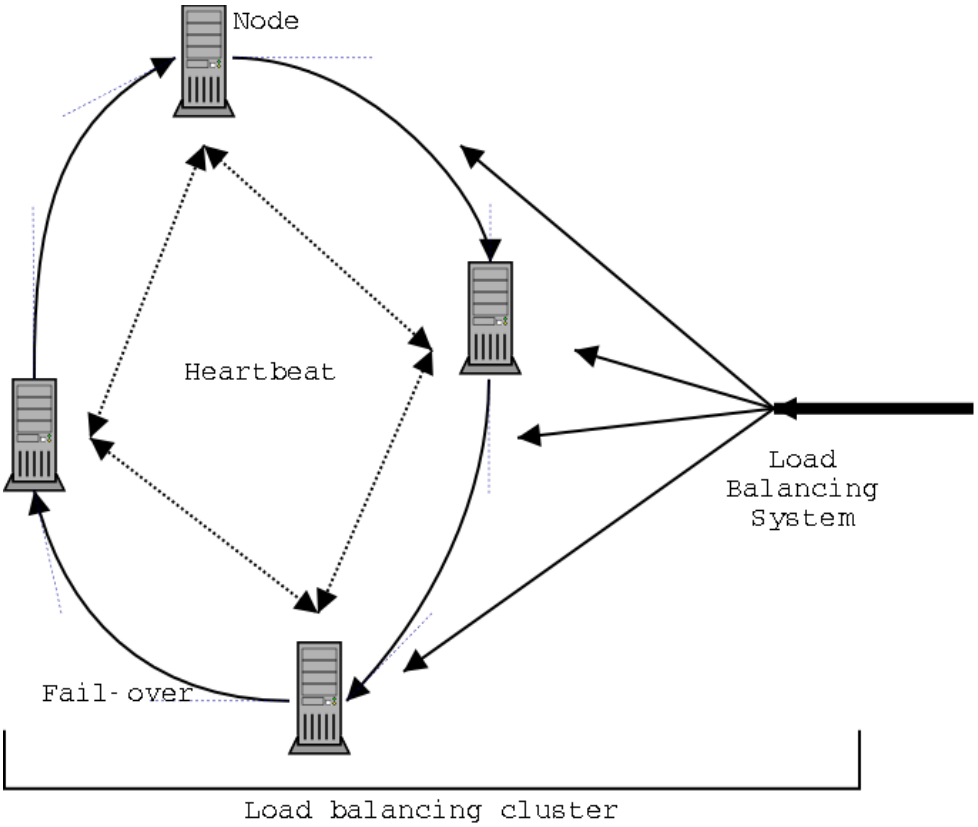

Figure 4. Components of a failure tolerant, load balanced cluster

The goal of our design is a system where the firewall consists of a number of near−identical, equal nodes. Spreading the load among nodes should take place independently from firewalling − allowing the use of conventional firewall systems. Let us therefore first consider the problem of load−balancing a firewall.

4.2.1. Load−balancing

In Section 9 we discuss a number of load−balancing techniques. Broadly, these break down into probe−based systems, and distribution−point systems. Probe−based systems require client behaviour − contradicting our requirement for no client modifications. Distribution−point systems are possible, but have a number of complications in the context of balancing network gateways. Firstly, a distribution point is a single point of failure. Secondly, firewalls connect multiple networks − so any distribution system must be present on every network, and must maintain concurrency between all distribution points. This is the only manner in which such a system could transparently ensure that both incoming and outgoing traffic on a connection passes through the same node.

An alternative method of handling load−balancing is to delegate the responsibility to the nodes themselves. Our approach is to deliver all incoming requests to every node. Each node would then handle a portion of that load (refer to Section 12 for details), statically defined by an administrator or agreed upon between nodes. This solves the issue of multiple connected network segments: each node sees all relevant traffic. One disadvantage is that this implies that the entire system is still constrained by the bandwidth available to individual nodes. However, since this would be the case for any distribution system, this is not considered a major issue.

4.2.2. Fail−over

No load−balancing system would be complete without support for handling node failure. In order to handle node failures, two systems need to be extended to support this: the load−balancing system itself, and optionally, the application level software. Without application−level support for fail−over, a node failure would sever any connections making use of that node, but would allow connections to be re−established. However, in order to allow the load−balancing system to shift load between nodes without interrupting connections, it relies implicitly on a suitable fail−over mechanism to avoid service disruption.

Therefore, in our composite system, application−level fail−over is critical. Our prototype, presented in Section 7, makes use of a specially designed application proxy, with periodic automatic state snapshots. In order to recreate a failed proxy, a new proxy is spawned with an old snapshot file as input. It uses this to recreate socket connections (hijacking the dead ones, see Section 8 for details), and reload its internal state to that of the latest snapshot.

Fail−over in the load−balancing system is described in more detail in Section 12. Once a node failure is detected, all live nodes are notified and take over the load previously handled by the dead node. Since the load−balancing system includes a facility for nominating backup handlers for node traffic, and for balancing these backups, the system is immediately balanced. The new handler nodes then nominate new backup nodes for rules, returning the system to a fail−over capable state.

4.2.3. Heartbeat

Finally, the firewall requires some system to track live nodes and detect node failure. Section 10 describes a number of heartbeat mechanisms, and Section 4.2.3 discusses

methods for handling heartbeat monitoring for a larger number of nodes.

Each node is tracked by one or more other nodes. When a node fails, a supervisor node will broadcast a failure notification to all live nodes. These will then proceed to take over the vacant load and stabalize the system.

Typical heart mechanisms ([MON], [Pirahna]) use master to slave connections. Rainwall [RAIN] uses a chained mechanism where tokens traverse all active nodes. Extensions to these schemes include mutliple connections between nodes (for example, where each node tracks both adjacent nodes in a circular order), and broadcast tracking. In broadcast tracking, each node track all other nodes using broadcast notifications. Due to the topology of this thesis' network design (see Section 10), this is a natural match.

4.3. Result

The result of adding a load−balancing system, a fail−over system, and a failure detection system to a firewall is a clustered firewall capable of running across multiple physical machines transparently.

5. Firewall Basics

In essence, a network firewall is a modified network gateway with advanced security features. It acts as a filter, controlling access to internal and external services. Firewall access controls can take many forms, ranging from simple access control lists (ACLs) to full proxy systems, where the firewall intercepts requests and handles them itself. [FWRFC] [Hunt−FW]

In this section, we will briefly cover some of the main aspects of firewalling: firewalled network topologies, filtering techniques, and extended firewall functions.

5.1. Firewall network topology

Firewalls control traffic passing through them − forwarding, modifying or blocking requests. In order to be effective, a firewall has to be present on every routing path between internal networks (which have a higher level of privilege and must be protected) and external networks (which have a lower level of trust and must be protected against). To paraphrase: a network is only as secure as the least secure system that has unrestricted access to it.

Irrespective of the internal configuration, a firewall can be considered as a simple network device with a number of interfaces (typically to trusted, external and demilitarised zone (DMZ) networks), which acts as a router with filtering capabilities. The actual firewall may, in fact, be a single router, or a network of routers, bastion gateways and proxies.

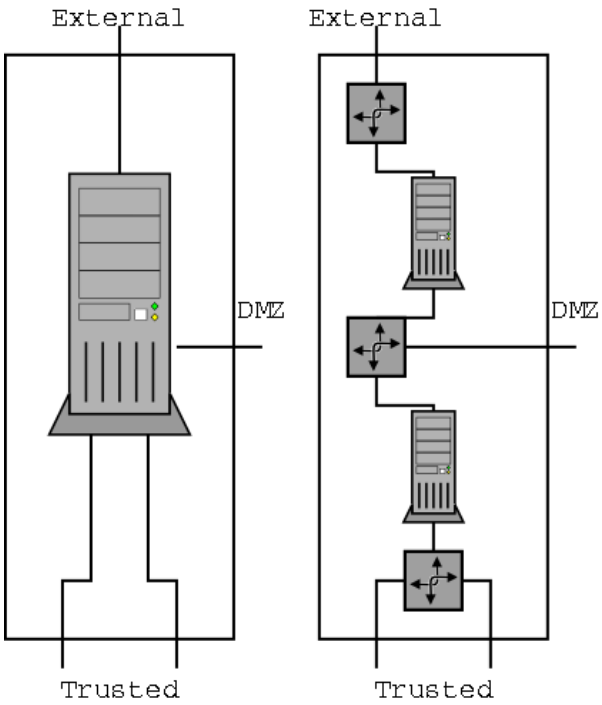

Figure 5. Firewall conceptual and a possible actual architecture

A feature found in many modern firewalls is the so−called demilitarised zone (DMZ). This is a network used to separate public services, which are subject to attack and possible compromise, from critical internal systems. Conceptually, this network behaves as illustrated in Figure 5 −where links to both the internal and external networks have controls in place.

A wide variety of firewall architectures is possible (see Chapter 6 of [Zwicky]), but most commercial configurations correspond closely to the conceptual firewall architecture: a single device with multiple network interfaces, some routing capabilities, and security software or firmware in place. In the remainder of this thesis, we will use this architecture as our basis.

5.2. Firewalling techniques

5.2.1. Access Control

The simplest firewall is simply a router with access control lists (ACLs) present, limiting the addresses that are dropped during routing. There are two problems with this technique, addressed by more advanced methods:

1. Filtering based on IP addressing is vulnerable to spoofing attacks.
2. Simple address−based filtering allows minimal traffic control. In particular, it is

impossible to create asymmetric access rules (differentiating incoming and outgoing connections), or to implement content filtering.

5.2.2. Static Packet Filtering

Static packet filtering, or simply packet filtering, determines whether to route or drop packets based on network protocol headers. In particular, the information about a packet commonly available includes:

- IP source and destination address.
- Protocol.
- TCP or UDP source and destination port.
- TCP flags.
- ICMP type.
- Interface received on, and interface expected to resend on.
- IP fragmentation flags.

Based on this information, it is possible to create service−specific filtering rules (for example, allowing only traffic that appears to be HTTP), and to create asymmetric rules for TCP. Allowing only outgoing TCP connections, for example, requires a rule that will allow packets with TCP SYN flags (without ACK) only when received on an outgoing interface. Connection setup request (SYN) packets from external hosts are dropped.

Packet filtering has problems handling complex (dynamic port or multi−connection) protocols. Often, supporting these protocols requires allowing access to large port ranges, potentially enabling access to unexpected services [PktFilter]. In addition, packet filtering has no content filtering capabilities, and may allow network−level attacks through.

On the other hand, packet filtering may be sufficient for low−security applications to simple protocols. Since only packet headers are considered, and no memory of previous packets is kept, packet filtering has a fixed low overhead. An efficient initial packet filter may, in fact, increase the capacity of additional, higher−level packet handling systems.

A good example of a packet filtering system is the Linux kernel version 2.2 *ipchains* system [IPChains].

5.2.3. Stateful Packet Filtering

Stateful packet filtering is a packet filter where the determination of a packet's fate also depends on information extracted from previous packets. In particular, stateful packet filters buffer packet data (while forwarding the packets) to extract details such as RPC portmapper response port numbers, and FTP PORT or PASV commands. The packet filter can then allow traffic to the appropriate port (for example, allowing traffic to port 20 on an FTP server after a PASV command), where it would otherwise not do so.

This connection tracking behaviour allows stateful packet filters to offer far smaller continuous exposure to protected services than static packet filters. In addition, administering stateful packet filtering configurations can be much simpler, due to the inherent understanding of application protocols implied.

The major problem with stateful packet filtering lies in tracking complex protocols. A stateful packet filter needs to understand every protocol tracked − requiring extensions of the base system to support new protocols. In addition, stateful packet filtering adds additional overhead in the form of packet processing time and connection tracking memory. However, while capable of blocking many of the stealth mapping techniques described in Section 3.2.2, it may still allow network−level attacks through.

Nonetheless, stateful packet filtering forms one of the staple techniques in modern firewalls due to its ability (though requiring special modules) to support complex protocols and the low overheads involved relative to full proxies. It also does not disturb the packet flow between source and destination. Finally, stateful packet filters can create virtual connections for connectionless protocols, allowing asymmetric filtering of protocols such as DNS.

Many current firewall systems use some form of stateful packet filtering, including the Checkpoint FW−1 firewall [FW−1] and the Linux kernel version 2.4 Netfilter system [Netfilter].

5.2.4. Network Address Translation

Network address translation (NAT) is more fully discussed in Section 2.1.5. From a firewalling point of view, a NAT system implicitly offers capabilities closely matching those of stateful packet filtering systems.

Since NAT hides network structures, mapping systems protected by NAT is

effectively limited to currently exposed hosts. For static and address−pool NAT, hosts (or addresses) that are reachable from external hosts are subject to mapping and attack. Port−pool dynamic NAT effectively blocks all access to internal hosts.

As with stateful packet filters, NAT requires additional processing overhead and protocol−aware modules. Externally inaccessible hosts are effectively protected from all external attacks (but may still be subject to trojan attacks), while the exposure of accessible hosts is greatly reduced. Network attacks on accessible hosts may still penetrate the firewall, and application−level attacks are unaffected.

5.2.5. Proxies

In general, a network proxy is an application that receives requests from clients, and processes those requests on their behalf − if necessary translating requests and contacting external servers [ProxyRFC]. Two common examples of network proxies include DNS caches and HTTP caching proxies.

In firewall proxies, the emphasis shifts from improving performance to increasing security. Typical functions of a firewall proxy include:

- **Hiding internal addresses** −by requesting information on behalf of internal clients, their details are hidden from external servers.
- **Eliminating internal information** − a proxy may remove or rewrite parts of requests to prevent sensitive information, such as internal host names or software versions, from being exposed to external hosts.
- **Restricting protocol functionality** − a proxy may block commands that do not comply with local security policies, such as filtering out HTTP cookies, FTP EXEC commands, or dynamic content in HTML pages. Depending on what parts of the data stream are modified, this is also known as content filtering.

In a pure proxy firewall, information is only allowed through when carried by a proxy process. This eliminates network level attacks and mapping of undisclosed resources. A correctly configured proxy is capable of any of the functionality present in NAT, stateful or packet filtering systems.

Firewall proxies suffer from two significant problems: performance and protocol support. Since a proxy acts on the application level, all data must be received, delivered to the application, processed, repacketed and transmitted. Cut−through proxies make use of kernel packet rewriting to transfer data between connections [TCPOpt], removing most of the overhead involved, but cannot be combined with

content filtering. In general, the use of proxies involves relatively large overheads.

The second problem lies in protocol support. Early proxies required protocol support (for example HTTP) or modified software tunnelling higher−level protocols (SOCKS [SOCKS]). Transparent proxies use operating system redirection functions to intercept requests not directed at the proxy, reducing the need for client support.

To fully utilise the power of proxies, however, the proxy software must fully support the specific protocol used. This, combined with the advanced functions implemented in proxies, makes the implementation of proxies complex and sensitive to protocol changes.

A good example of a proxy firewall is the TIS firewall toolkit [FWTK]. Many commercial firewalls also include proxies − refer to [FWList] for details.

5.3. Extended Firewall Functionality

By virtue of their position in network topologies, firewalls are also natural locations to implement security−related functions other than access controls. In particular, some functions commonly found in firewalls include:

- **NAT** − While NAT can be used as a form of access control, it also makes sense to place NAT functionality on the firewall. Firstly, the firewall already forms a routing bottleneck to the network. Second, by applying NAT inside a firewall network, the construction of address−based rules is made much more difficult − since the rules would have to act on translated rather than actual addresses.
- **Accounting** − In many networks, there is a need monitor use of network resources. By virtue of its position and existing logging capabilities, the firewall forms a natural point at which to implement resource tracking.
- **Virtual Private Networking** (VPN) − Virtual private networking is a technique used to simulate the access privileges of local users for remote hosts. Typically, this uses cryptographic tunnelling protocols such as PPTP [PPTP] or IPSec [IPSec]. By implementing VPN functions on the firewall, the intensive processing requirements of cryptography is offloaded onto the firewall itself, which can be optimised (or have hardware assistance) to address this. Similarly, configuration is limited to the firewall, reducing interoperability problems.

  There is also the problem of filtering encrypted traffic. Host−to−host tunnelling would make service or content based filtering impossible, negating the effectiveness of most advanced firewalling techniques.

- **Authentication** − A firewall, as the demarcation point between trusted and untrusted resources, is a natural location in which to place strong authentication mechanisms. This allows a single point of configuration, and guarantees a trusted authentication server[30]. This also allows filtering rules to be set according to authenticated user identities, as opposed to local addressing − supporting situations where users may move between network hosts.

  Many protocols do not inherently support authentication, and maintaining the same identity between different protocols automatically may not be possible. One common technique is to associate the local address of an authenticated user with any rules based on that identity. This allows all requests from that address to be identified as that user's, but presents problems in multi−user systems, multi−homed hosts, and may be subject to internal spoofing.

Another aspect developing in firewall applications is the concept of personalised firewalls. Here, every host has a small firewall running in its network stack − in effect, combining the advantages of hardened hosts and firewalls. This approach also offers the ability to implement process−specific control of outbound connections, reducing the risk posed by trojan software. Refer to [DistFW] and [PersFW] for more detail.

30 The firewall may implement authentication itself, or act as a proxy to another server. Authentication typically takes the form of a password or token−key, using protocols such as RADIUS.

6. Problems with conventional security

6.1. Firewalls

Modern firewalls suffer from two basic problems: increased bandwidth and attack complexity. As has been described, conventional monolithic firewall architectures are unable to scale well to serve high−bandwidth connections.

Firewalls have been shown to be effective in controlling access to private services, using filtering techniques. However, in order to be efficient, filtering firewalls attempt to minimize the modifications made to incoming data − typically only performing minor packet readdressing, some sanity checking, and accept/deny filtering. This behaviour means that any packet−level attacks, where the packets in themselves are valid, can pass through a firewall. One example of such an attack is WinNuke [WinNuke], where TCP priority data is passed to a public NetBIOS service. While the packet is valid within the TCP protocol, the NetBIOS server could not handle priority data and crashed.

This behaviour can trivially be extended to application−level attacks (described in Section 3.2.3) and firewall packet filters and proxies. In the case of packet filters, application−level attacks are unobserved by the firewall, and so pass through as valid requests. Application proxies, while capable of observing application level attacks, often cannot distinguish valid requests from attacks without application−specific knowledge. Again, in order to improve performance and maintain service semantics, the attack is passed through with minimal modification.

In short, a general firewall will allow any attack that is valid in its transport protocol to pass to protected hosts. In other words, a firewall offers limited protocol validation (the quality of which depends on the specific implementation), but does not offer protection against valid misuse of protocols. For example, a firewall could distinguish a valid HTTP request from a binary stream, but could not recognise a request containing dangerous URLs (refer to Section 3.2.3 for more detail on attacks).

6.2. Cryptography

Cryptography is comprised of of a number of techniques to ensure the privacy and integrity of data. The use of a cryptographic protocol generally insures one or more of:

1. The data sent was not observed by unauthorised parties while in transit

2. The data has not been modified while in transit
3. The data was sent by the expected remote peer

Again, cryptographic techniques offer no remedy to data that was inherently flawed − by choice or by accident. In the case of cryptographic techniques that do not authenticate the remote party (such as HTTPS), or in the case of untrusted (though authenticated) peers, the data could represent an attack. This method allows attackers to bypass other security systems (notable IDS), while retaining access to vulnerable services.[31]

In essence, cryptographic communications are only as trustworthy as the remote partner − using cryptography to communicate with anonymous hosts does not improve local security, and may negate other security mechanisms (IDS, intelligent proxies, etc.).

6.3. Intrusion Detection Systems

Intrusion detection systems, like firewalls, suffer from a number of problems: increased bandwidth, attack complexity and volatility, lack of remedial mechanisms, and problems in predicting behaviour of the underlying and remote network systems. Increased bandwidth is being addressed by decomposed ID structures [GrIDS], hierarchic reporting chains [AAFID] [Emerald], and the scope of inspection being limted to critical zones.

Resolving attack complexity and volatility is the object of a number of ID research projects, including complex attack models [Kumar], honeytrap systems for learning attack behaviour [Honeynet], heuristic attack recognition, and anomaly detection systems [NIDES] These systems attempt to increase the intelligence of attack recognition in IDS, reducing the reliance on comprehensive signature libraries.

Ptachek [Ptachek] and *whisker* [whisker] present a number of techniques available to invalidate the output of ID systems. While some of these techniques can be addressed by improving IDS implementations, resolving others would require extensive knowledge of the local client and network behaviour.

Finally, IDS is inherently an observational facility; in most cases, current systems have purely responsive capabilities. An IDS is usually incapable of preventing an attack from proceeding − at best, it can attempt to mitigate the effects of that attack or

31 As was demonstrated in the case of the PCWeek hackers' challenge [HTTPSHack], where an attacker completely bypassed IDS systems by using an HTTP attack tunnelled through HTTPS.

prevent future attacks. This follows from the fact that an IDS recognises side effects of an attack (log entries, or network traces observed at the same time as the victim). In order to be effective against attacks, an IDS would have to be capable of recognising them before they take effect.

6.4. Silver bullet

This thesis is targeted at mitigating the effect of many of these problems. Our proposed distributed, proxying firewall with ID capabilities built into proxies would have the following advantages:

1. The distributed architecture makes handling high traffic bandwidth possible. Since the system should allow virtually unlimited processing capacity, bandwidth would be limited by inherent device capacities.
2. The use of application proxies can virtually eliminate the effects of lower−level attack techniques. Such attacks will be intercepted by hardened firewall nodes, and any data payload repackaged in safe datagrams.
3. Application proxies, combined with ID attack recognition methods, allow incoming requests to be verified before leaving the firewall, eliminating known attacks before they take effect.
4. The use of proxy flow control, data buffering, stream reassembly and decoding methods unifies the data observed by IDS and the protected endpoint, resolving interpretation ambiguities. This can allow the IDS to detect obscured attacks.

Contrary to the section heading, this will not solve every problem − and may introduce other issues − but it would significantly enhance the function of a separate IDS / firewall system.

7. Design and implementation of a Transparent Proxy

The basic design of our firewall relies on two techniques: stateless packet filtering, and application proxies[32]. Stateless packet filtering is extremely fast, simple to configure and adapt for new protocols, but cannot support security generically for multiple connection protocols, and does not offer fine−grained controls. Proxies suffer from performance issues − all information must pass through the protocol stack twice − but offer almost unlimited flexibility in function, and can be developed using general network service techniques. We believe that these techniques represent the two extremes of firewalling, and combine to offer high performance and exceptional control for complex protocols.

In this thesis, we assume the use of an external packet filtering system (such as the Linux *ipchains* [IPChains] or Netfilter [Netfilter] systems), which may in fact be stateful or stateless. The fundamental requirement is that such a packet filter supports simple protocols: in the remainder of this section, we present the transparent proxy structure we have developed to support complex protocols.

7.1. Requirements

In developing the proxy framework, there are a number of basic requirements:

1. The framework must be capable of transparently[33] tunnelling TCP traffic.
2. The framework must support transparent proxying, request redirection, subsidiary proxies, and other features found in existing proxy applications.
3. The framework must have a secure basic design. It must, by design, be insensitive to typical network attacks.
4. The framework must be extensible to support a variety of network protocols and functions. In this thesis, we have implemented protocol−neutral (also known as circuit−level) and HTTP transparent proxies.
5. The framework must support the fail−over and resumption techniques developed in this thesis. As part of this, the framework must maintain its internal state and connection buffers in such a manner as to minimize data loss on proxy failure.
6. The framework must have reasonable performance.

32 The third major firewalling technique, stateful filtering, is a compromise between these methods. In addition, fail−over of such systems has already been commercially implemented [CHA], and could be done using the same methods we develop here for proxies.

33 Without requiring client or protocol support, ideally without being visible to clients or servers.

7.2. Our framework

A proxy is an application that receives requests, processes them (generally by contacting a real server), and returns results. Proxies are commonly used to enhance the functionality of clients, servers or protocols − for example, by offering improved caching, load−balancing, or better security. The framework we describe here is an implementation of the functionality common between proxies for a variety of protocols. In particular, we will be discussing implementations of protocol−neutral and HTTP proxies with reference to this framework.

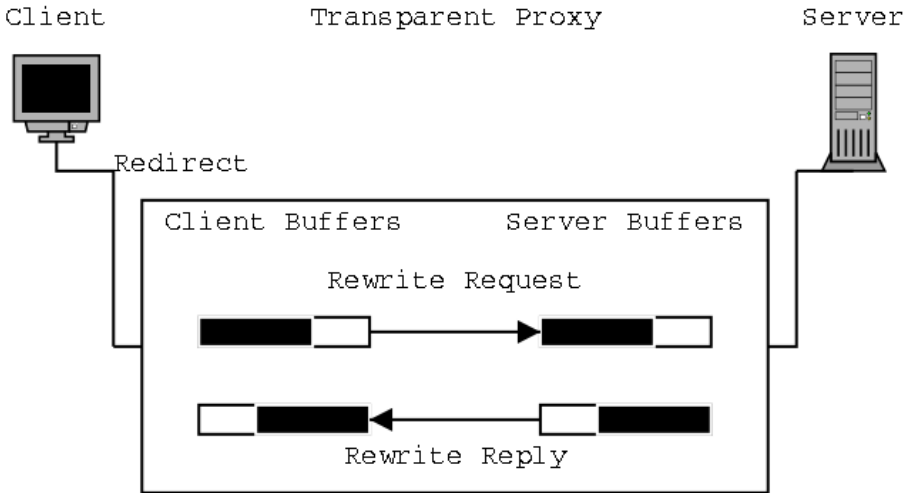

Figure 6. Structure of a transparent proxy

To place our discussion into context, consider the one possible life cycle of our protocol−neutral proxy (Figure 6):

1. An administrator sets up redirection rules to force all HTTP requests passing through our firewall to be rerouted to local port 80, using Netfilter.
2. The administrator also starts up the proxy, with configuration options set to redirect traffic on port 80 to a local web cache (*wwwproxy.canterbury.ac.nz*).
3. While initialising, our framework executes first. It starts by allocating memory for the persistent internal state, parsing configuration options, setting up logging, and placing itself into the background. The proxy is now running as a daemon.
4. The proxy waits for incoming requests. On receiving a request, it forks off a child process and returns to wait for further work.
5. The child process first inspects the incoming request, determining its original destination. For a transparent proxy, some incoming requests were originally destined for other servers − the address of which often cannot be derived from the request content.

6. The child proxy opens a connection (to *wwwproxy*, or, if that fails, to the original destination), and calls a protocol−specific connection handler.
7. At this point, control passes from the framework code to a protocol handler, requiring a the latter to support the framework's data flow methodology. The protocol−specific handler receives newly opened TCP connections to the client, and some remote server. It has the responsibility to handle any traffic on these connections.
8. The protocol handler prepares its internal buffers, and enters an input−driven loop. When any connection has data ready to read, as much as possible is read into internal buffers. When any connection is ready to write (and there is pending data), output is sent to the kernel for transmission.

   Once data is received, how it is handled depends on the specific protocol. In the protocol−neutral case, data is simply passed from the client read buffer to the server write buffer (and vice versa).[34]

   In the case of specific protocols, such as HTTP or FTP, input is buffered until a complete request or command is received[35]. The whole chunk is then rewritten (stripping out unacceptable parameters or content), or dropped, and the result transferred to an output buffer. In addition, the nature of the request, and whether it was dropped, is pushed onto a local stack − allowing injection of appropriate results for commands that were disallowed.[36]

   After every cycle, the proxy process requests an asynchronous update of the persistent state file image[37]. It then returns to sleep until new input arrives or buffered data can be written.
9. This loop continues until a connection closes. Once data is fully flushed to the matching output connection, that connection is also closed. At this time the proxy terminates, returning control to our framework.
10. The framework cleans up any open connections, frees up and removes persistent

34 A special case is handling TCP urgent data. The standard sockets API [Stevens−1] does not fully support this TCP function, so our proxy framework strips out or forwards TCP urgent data as presented by the sockets interface. This may change the flow of urgent data, but should be indistinguishable to sockets−based clients and servers.

35 This requires sufficient buffering to allow the processing to match inherent protocol granularities − such as individual FTP commands or HTTP requests − eliminating problems with inconsistencies due to previously−passed data.

36 Notice that there is no concept of locking here. Since system failures do no honour semaphores, the best we can do is minimize the critical area and attempt to recover if necessary.

37 We use memory−mapped files, delegating synchronisation of the internal state and disk files to the operating system.

state, and terminates.

A proxy that is interrupted while outside its input loop does not yet have a cohesive state, and cannot be resumed. This is acceptable, since this will only occur at the start (when a connection can simply be restarted), or at the end (when handling is already complete) of a request. If interrupted while in the input loop, another proxy can resume processing using the state file:

1. Proxy starts up, with the deceased process' state file as input. It initialises as usual, but does not daemonise.
2. The proxy then replaces its internal state image with that loaded from the state file. This imports buffer contents, connection information, etc. − anything that the deceased proxy had stored.
3. The proxy reconnects to the client and server by hijacking the lapsed connections (see Section 8 for details on the hijack process). It restarts the protocol−specific request handler input loop, bypassing the usual state initialisation.
4. From this point onwards, processing proceeds identically to what would have happened if the proxy had not been interrupted (barring data loss during the resumption process, refer to Section 7.4).

Should the proxy be interrupted again, this process repeats. The only real limitations are data loss during resumption, the requirement that a recent state file be present, and lack of support in the current *prism* implementation (see Section 8.4) for overlapping connection hijacks. The latter is not considered a significant problem, since proxy failures are expected to be rare, and generally to coincide with full host system failure − where a different *prism* instance would handle resumption.

7.3. Fail−over in action

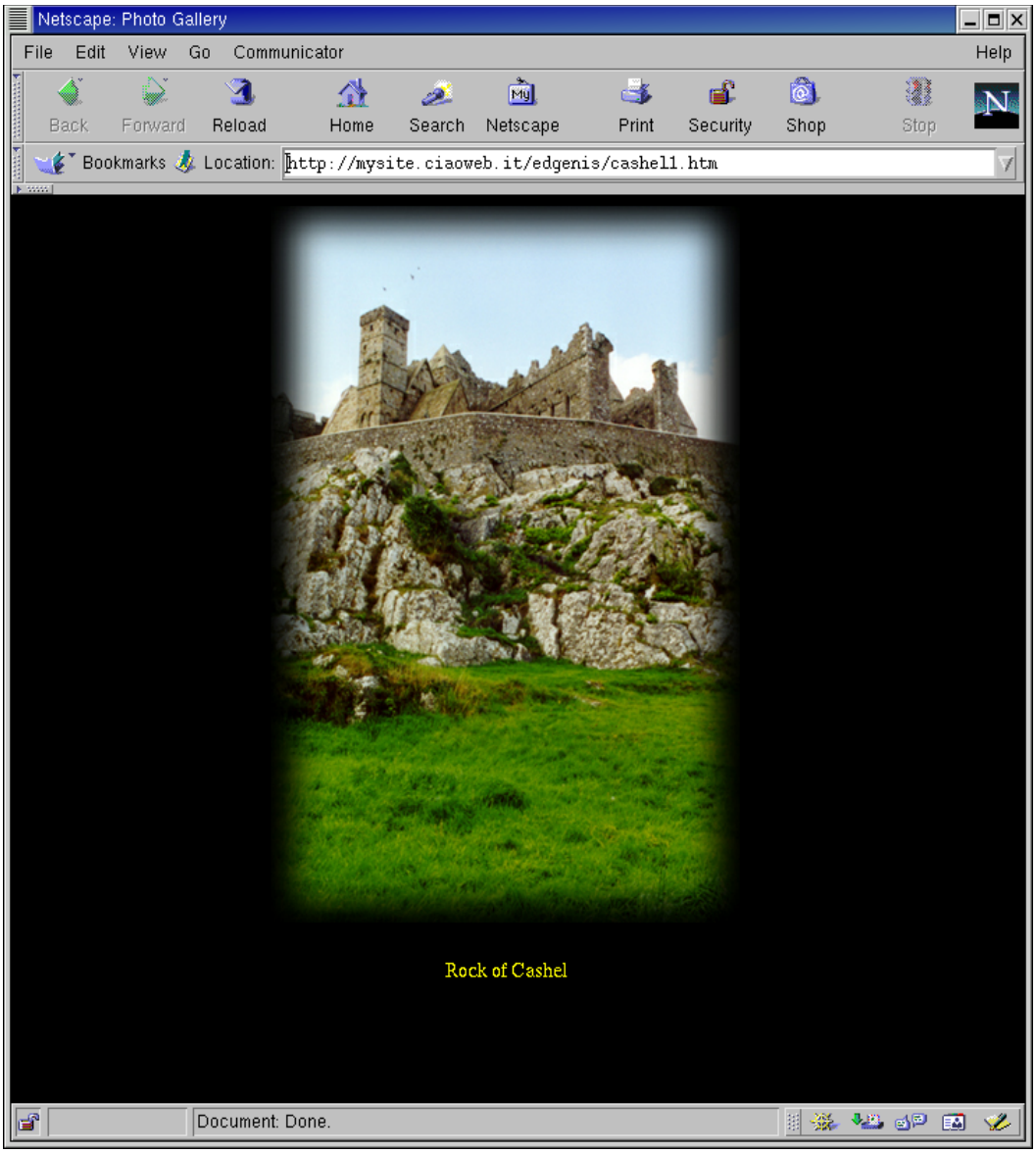

Figure 7. A normal page load via the HTTP proxy

Figure 7 shows a web page [Castle] loaded via our HTTP proxy implementation, without interruption. The proxy log for this and the following session is included in Appendix 2.Should the proxy be interrupted while loading the image file, the resumption process outlined in Section 7.2 is invoked.

Figure 8 shows the result. In this case, the point at which the transfer was interrupted can clearly be seen as a disruption in the middle of the image. The interruption, in the test system, takes the form of a software signal from a different process terminating the old proxy. Since the operating system sockets for that proxy remained functional, data from the web server continued to be acknowledged, leading

to the extended data loss[38] (shown at the bottom of the loaded image) visible in this case. Once a new proxy has resumed (in particular, once *prism* has relinked the affected connections), data flows into the new set of sockets, and the remainder of the image loads. The only form of communication between the old and new proxy versions is a file containing the persistent proxy state.

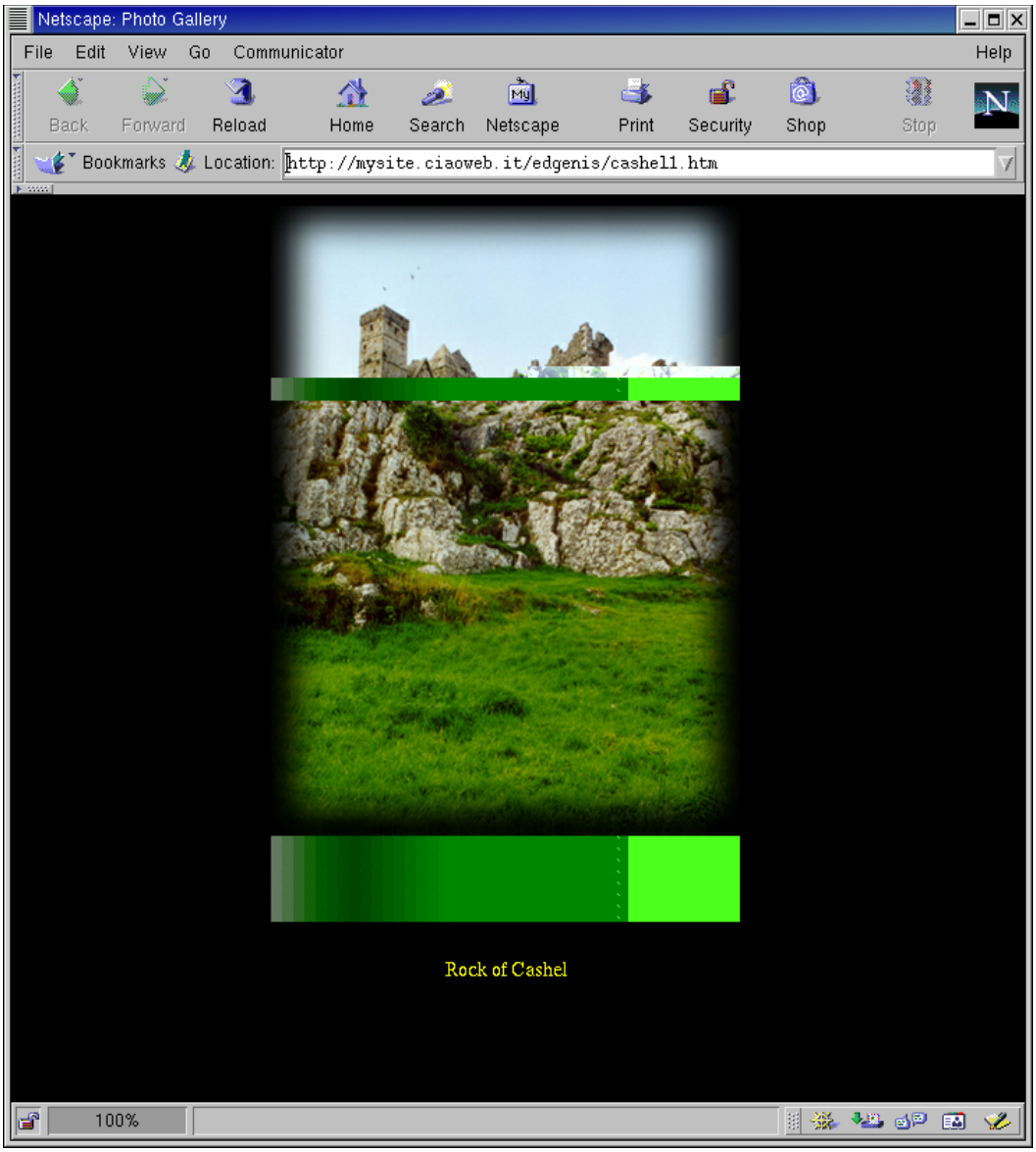

Figure 8. An interrupted page load demonstrating the HTTP proxy resume process

This demonstrates that the resumption, while leading to data loss on active connections, is feasible. In Section 7.5, mechanisms for reducing the disruption in data flow caused by proxy replacement are discussed.

38 In a real implementation, the host containing the failed TCP endpoint would also be dead, limiting data loss to one TCP window size plus any unsaved data in the proxy buffers.

7.4. Data loss during resumption

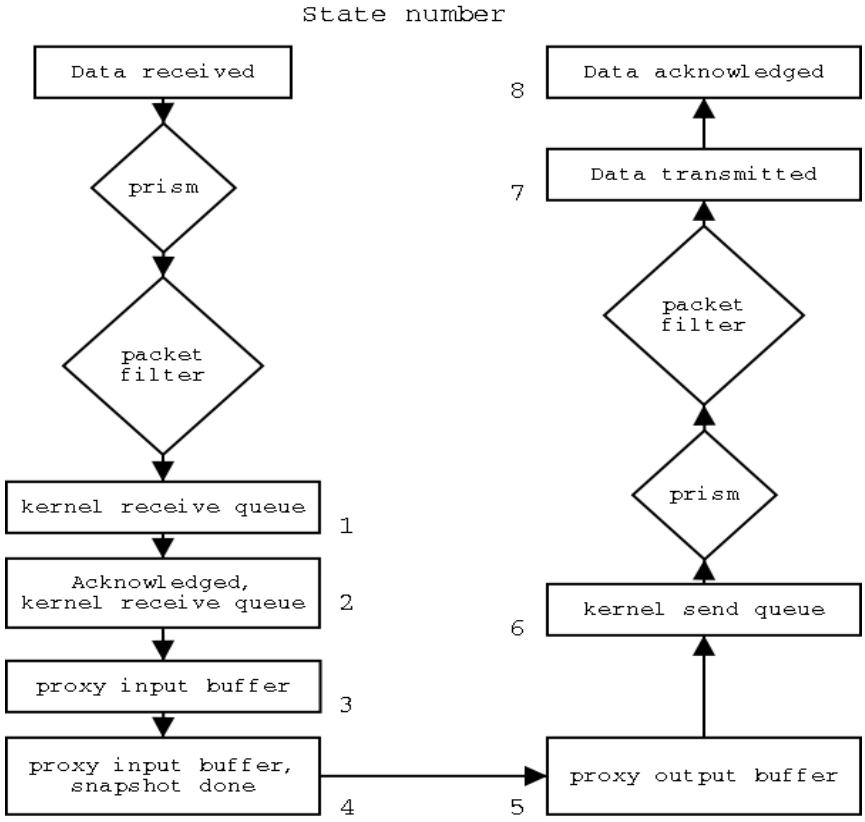

Figure 9. Stages in proxy data handling

Figure 9 outlines the stages observed by data while traversing the proxy firewalling system. Data is received by the TCP stack, acknowledged, queued in the kernel and proxy, sent back out via the TCP stack, and remotely acknowledged. In this section, we will discuss what happens when the proxy fails (and has data pending at various stages of this cycle), and how this affects the design we have outlined.

Firstly, we assume that data saved in a snapshot is safely held against proxy failure; anything else is at risk. Secondly, we assume that proxy and node failures coincide in general; though we will discuss shortly the effect of proxy failure without node failure. At the point of proxy failure, different parts of the request data stream will be in different states of the outlined life cycle. Consider the effect of proxy failure on data held in each of the numbered states:

1. Data that has been received by the kernel but not acknowledged. When the proxy (and node TCP stack) fails, no acknowledgement will be sent − causing the remote endpoint to resend this information. Since this data is also absent from any snapshot, recovery will not duplicate this. Therefore, this data is held safely by the

remote sending endpoint.

2. Data that has been received and acknowledged. The remote endpoint will discard local copies of this data once acknowledgement is received (and there is no mechanism for retracting sent acknowledgements). This data is also absent from any snapshots. Therefore, data in this state cannot be recovered.
3. As with data in State 2, this data is neither held remotely nor safely stored. Therefore, this data is lost on proxy failure.
4. Under our initial assumptions, data in this state is recoverable. Another node can use the snapshot to recreate a proxy holding this data.
5. This data is either present in the most recent snapshot as buffered output, or as unprocessed input. In the latter case, it will be regenerated on proxy resumption.
6. This data is no longer held in snapshots, nor is it held by any remote host. Therefore this data could not be recovered
7. Data transmitted but not acknowledged may be received by the destination endpoint. If it is not received, this data falls into the same category as that of State 6, and cannot be recovered. If it is received, the remote endpoint will send an acknowledgement independently from any local failures, reducing to State 8.
8. Data sent and acknowledged is safely held by the destination endpoint, and does not require recovery.

Therefore, it can readily be seen that any data held in kernel queues is subject to loss. One possible approach to resolving this problem is to merge the kernel and proxy buffers − modifying the kernel to use process buffers. In that case, States 1, 2 and 3, and States 5 and 6 would be unified. Data would only be acknowledged once placed into proxy buffers, and would only be removed once correctly acknowledged by a destination endpoint.

A more fundamental problem lies in the potential data losses in States 3 and 5 − data held in proxy buffers but not saved in snapshots, or data that has been delivered but is still held in snapshots. In the first instance, this leads to data loss, in the second, to data duplication. In the next section, we outline potential solutions to this problem, but since this would require significant modification to both the system kernel, and to the sockets programming interface, we have not implemented these in this thesis.

7.5. Methods for resolving data loss

7.5.1. Proxy flow control

The solution to data loss during resumption, in essence, lies in allowing tighter flow control to the proxy application. In the conventional sockets−based system, any data queued for delivery in the kernel is handled transparently − flow control is entirely within kernel space. By shifting this control to the proxy, which is also responsible for guaranteeing data recoverability, it becomes possible to offer TCP delivery guarantees spanning the entire connection between remote endpoints.

For this to work, the following modifications are necessary:

1. The proxy should be able to extract the correct TCP sequence number for any given byte of data from the system. This allows the recovery process to estimate the size of data disruption, and the corrective action needed on recovery. (Therefore, if the snapshot contains bytes 1000−1500, and the remote host has received up to byte 1300, the proxy should avoid resending (or at least use correct sequence numbers) for bytes 1000−1300.)
2. Data should not be acknowledged by the operating system until safely received by the proxy − a superset of the solution described above. In the conventional sockets interface, the kernel acknowledges any data stored in local buffers. Replacing this behaviour with a system call that allows up to a given sequence number (or a given number of bytes) to be acknowledged allows the proxy to maintain data in remote buffers until safely stored[39].

The first requirement allows the proxy to track data transfers, and avoids data duplication. The second requirement allows the proxy to avoid data loss, at a possible cost of TCP performance, due to the loss of TCP flow control features. Taken in combination, these allow better delivery guarantees in the face of process or node failure than can be offered by using only periodic snapshots.

7.5.2. Silent slaves

A common alternative approach to the same problem, is to run multiple instances of the software on different nodes. One instance is nominated as the master, while all

39 The effect of such an *advance()* call on TCP flow control algorithms bears further consideration, however, since proxy processing is limited by process scheduling and contention.

others are designated as slaves. Only the master instance can take action on a client request − all slaves interpret requests to update their internal state only. When the master fails, the slaves theoretically hold identical state to the master.

This approach has a number of complications. Firstly, slave processes require as much processing as the master − which, for applications where processing requests is computationally intense, can result in significant overhead[40]. Secondly, the behaviour of such processes must be completely deterministic, and the host environment must be sufficiently similar to guarantee identical results. In the case of TCP proxies, this would require modification to the TCP sequence number selection mechanism, or require proxies to compensate internally[41]. Similarly, local port selection, process scheduling disparities and many other environmental factors would need to be taken into account.

Another problem lies in handling multiple failures. In this approach, a slave must be observing a sufficient part of the connection life cycle to configure its state. In the case where proxy parameters are set up in the initial handshake, any slaves must be present for the entire life of the request. This implies that a finite number of connection failures can be handled − new slaves cannot be created for an existing request. Finally, unpredictable delivery of individual packets (which, again, tie into node conditions) creates the risk of desynchronisation between master and slaves.

In practice, this is, or appears to be, the technique in use in a number of commercial redundant systems [WGHA][CHA]. In all of these cases, however, the effect of node differences is minimized by limiting the system to simple redundant configurations. This is neither scalable nor makes efficient use of resources.

7.5.3. Buffered catch−up

Another approach we are considering lies in a combination of a snapshot and a tracking approach. The idea is to have a snapshot of the ’master’ proxy (ideally including sequence number tracking information, but without proxy flow control) and a buffer of recent incoming packets pertaining to that proxy. The resuming proxy can then use the combination of these to rebuild the proxy state from the point of the last snapshot.

40 The resulting processing load is the original load, multiplied by the number of slaves used − effectively reducing the available capacity by that factor.

41 This is another application for sequence number mapping as implemented in *prism*, avoiding the need for intrusive OS modification.

In order to use this method, a mechanism for extracting, buffering and reinjecting packets into the local TCP stack is necessary. In addition, the process supervising packet buffering should track the sequence number state of the master process as shown in snapshots − allowing the supervisor to discard packets already handled by the master.

Since the current Linux kernel already includes facilities for packet extraction and injection, this method would require minimal kernel modification. In fact, the only modifications necessary should be those allowing sequence number tracking in the proxy. This method resolves many of the problems inherent in either of the other approaches described above − it uses conventional TCP flow control, avoids the need for slave proxies, and allows multiple failures to be handled. However, this method can be subject to desynchronisation (when one node loses packets and another does not), and requires sufficient time for a new backup node to queue packets since the last snapshot before fail−over is possible.

8. TCP session hijacking: the light side

TCP is a connection−oriented protocol offering reliable, ordered byte stream transfer. Reliability is maintained by the use of synchronised byte counters at both endpoints. This synchronisation offers some protection against data injection and removal, as these are seen as link failures and corrected, unless an attacker uses the correct sequence numbers − a technique known as TCP session hijacking.

TCP session hijacking is an extremely powerful but difficult attack. In order to successfully execute a hijack, an attacker would have to correctly guess the appropriate sequence numbers out of the 2^32 possibilities − assuming secure TCP implementations are used. Note that a number of TCP/IP implementations use predictable initial sequence numbers for TCP connections, vastly simplifying the problem. Blind TCP hijacking typically consists of vast numbers of sequence number guesses − and can thus be recognised. See [TCPHijack] for more details on this technique.

A more dangerous problem is non−blind TCP session hijacking. If an attacker can observe a connection − for example, by controlling an intermediary router or a machine on the same broadcast segment[42] − he has full knowledge of the sequence numbers in use. In this case, the only effective defence against session hijacking is cryptography.

This feature of session hijacking can also be used to improve service, however. When a connection endpoint crashes, the other end of the connection remains intact for a short period. If we could recreate a half−connection endpoint locally to replace that lost in the crash, we may be able to avoid service interruption due to the crash. In effect, this is a special case of TCP hijacking − hijacking a connection where one endpoint is unresponsive.

In order to hijack a session, there are two fundamental requirements: the hijacking host needs to be on the connection path, and we need the correct sequence numbers for that connection. When creating replacement proxies, the first follows from the premise that all connections pass through the firewall, based on failure recovery in the load−balancing client. The second requirement is more difficult: we have implemented two mechanisms to extract this information.

42 Which includes switched networks, by using techniques such as switch MAC address cache flooding outlined in [Hunt].

8.1. Compliant client sequence number extraction

The first mechanism relies on a specially modified TCP/IP stack, and client software that stores sequence numbers. In order to restore an endpoint, we need to recreate a compatible process state − ideally, by recreating the state of the lost process. To do this, we require state information from that process.

Sequence number extraction works by requiring the client process to track its own sequence numbers, and store these as part of its internal state. When later restarted on a different system, this state can be directly used. This requires that the TCP/IP stack publish these numbers in some way, which is not generally the case. Therefore, this technique generally requires modifications to the TCP/IP module in the operating system (something which we strive to avoid where possible).

This technique guarantees a reasonable approximation of the correct sequence numbers, and is the only method for correlating sequence numbers with data sent by the client. Should the sequence numbers be out of date (as is the case when an appreciable amount of traffic has flowed through the connection since the most recent client sequence number check), these will not be directly useable. This can greatly complicate the mechanism for resuming connections.

8.2. On−demand probe sequence number extraction

The TCP protocol specifies that, when receiving a packet with an incorrect sequence number, the handler responds with an ACK packet specifying the correct numbers [TCP]. We use this feature as a form of probe, querying the surviving endpoint (which still maintains correct synchronisation) for sequence numbering. See Section 8.4.1 for implementation details.

Since we are on the data path for the failed connection, having replaced one endpoint, we will receive any response from the remote endpoint. There are two possible cases: first, that the remote station sends data, or second, that the local system attempts to send data. In the first case, we know that the sequence numbers are correct, so we simply use those numbers. In the second case, we do not have any sequence numbers to map the outgoing data into the remote synchronisation − so we simply send the data without any mapping. Unless the randomly chosen sequence numbers happen to fall inside the remote acceptance window (the probability of which is inversely proportional to the ease of hijacking the connection blindly, and is therefore unlikely), the remote host replies with a synchronising ACK. At this point, we simply

accept these numbers, mapping later packets accordingly − including an automatic retransmission of our initial probe.

This technique extracts sequence numbers as they are at the time of reconnection: any data in transit before this point is lost. Without buffering data independently from the deceased client process, this cannot be completely avoided (see Section 7.4). Allowing this approximation does, however, simplify the design of our client system a great deal, and has experimentally produced reasonable results.

8.3. External sequence tracking

A third approach that we have not implemented in this system, would be to have an external module that observes connections and independently tracks sequence numbers. Such a module would have to have access to all packets in tracked connections (introducing a significant processing overhead), and would have to emulate the TCP sequence number handling of the client system fully (complicating design). However, such a module would, when combined with client sequence number extraction, be able to estimate how much data has been transferred since the last client snapshot.

This approach is similar to that used in ID systems and connection tracking network devices (high−end stateful packet filters and NAT devices), and is subject to the desynchronisation problems described in [Ptachek].

8.4. Implementation of a TCP session hijack API

When a network client terminates abnormally (without cleaning up open network connections), the remote endpoint is temporarily unaffected. As long as it does not receive any indication to the contrary, it will continue to believe that the failed client is alive − until some timer expires. This allows us an opportunity to replace such a dead client without dropping a connection.

In order to replace a failed network client, one of the first steps lies in re−establishing any network connections. This is not as simple as reopening the connection, since the remote client believes that a connection already exists. Indeed, what needs to be done is to locally recreate a socket[43] compatible with the existing remote socket.

One method of doing this would be to use a modified TCP stack and operating system, which would allow direct injection of socket structures into internal tables.

43 A socket is the network equivalent of an open file: it represents an abstraction used by operating systems to mask network protocol complexities.

This approach, however, would require significant modifications to privileged systems −which greatly complicates implementation.

A more general approach is to use a custom module (named *prism*) that hooks into the Linux Netfilter framework [Netfilter]. Due to the excellent support for user extensions present in this system, no intrusive kernel modifications are necessary.

The simplest manner in which to think of *prism* is as a variety of NAT − except that, while NAT does address rewriting, our module also rewrites sequence numbers. Another application of this type of technique is in the creation of cut−through proxies [Pix][TCPOpt].

In general, *prism* consists of two simple phases: supporting the setup of a local TCP endpoint, and mapping packets between endpoints. To set up a local endpoint, the initialising client opens two sockets (one active and one passive), registers with the module, and attempts to connect to the remote endpoint. *prism* loops back the handshake into the passive local socket (thereby allowing the local TCP stack to take care of handshaking complications) until the connection is established, at which point it disconnects the two local endpoints. The passive endpoint then terminates, leaving a singular local endpoint matching the remote surviving endpoint.

Mapping proceeds by intercepting all packets between the local and remote endpoints, rewriting the sequence numbers to match the equivalent values for the other destination endpoint, and allowing the packets to be delivered normally. Sequence numbers used for this mapping are retrieved using the techniques described in Section 8.1−3.

8.4.1. Implementation details

In the Linux kernel version 2.4, the Netfilter framework allows external modules to process packets during their traversal of the networking stack. This allows a variety of packet processing tasks: connection tracking, packet filtering, and our hijack support module, among others. Our module will process IP packets fresh from the wire − then forward these on to the remainder of the networking stack.

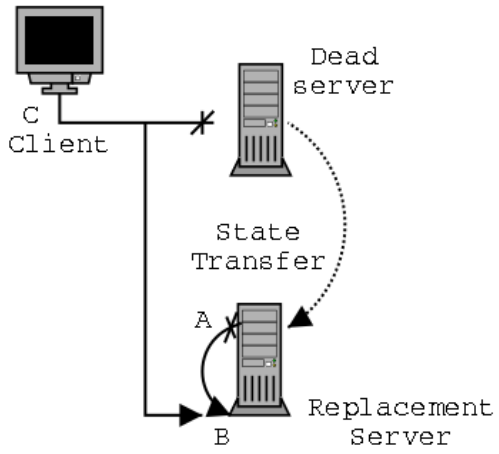

Figure 10. Prism server replacement outline

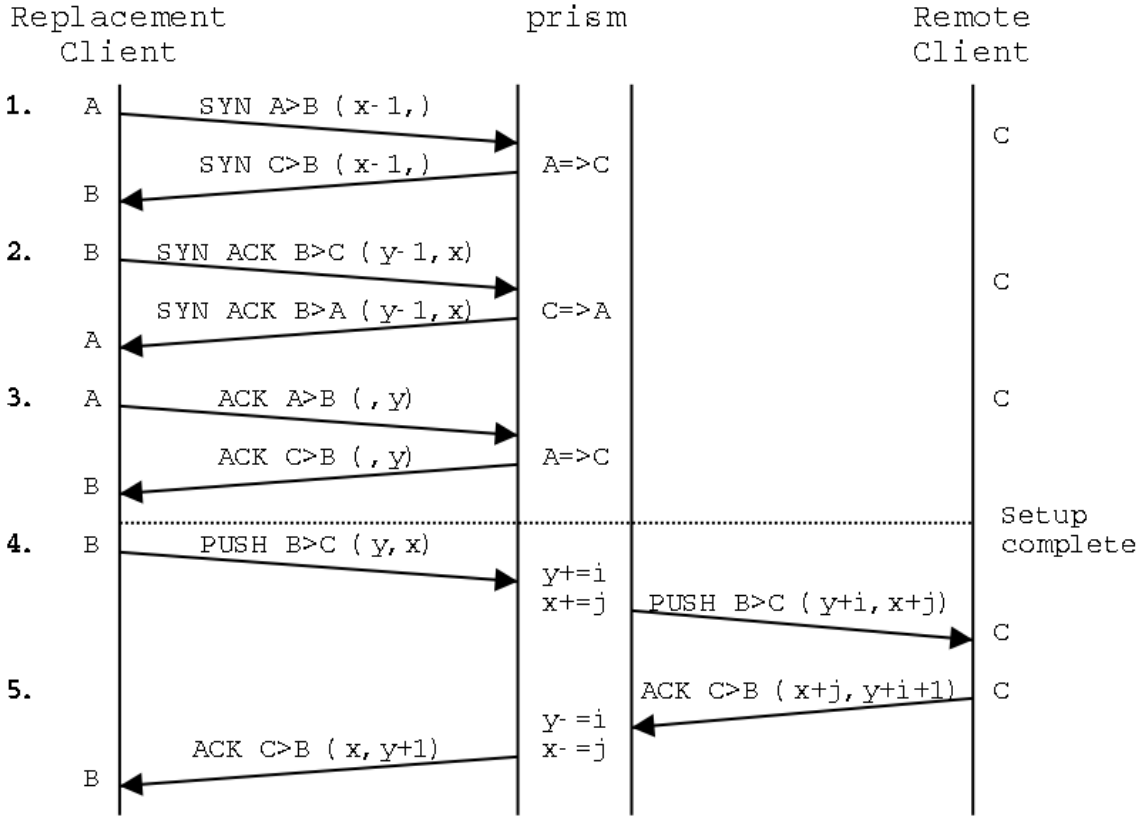

Figure 11. TCP connection reestablishment procedure

Refer to Figures 10 and 11 for a graphical summary of *prism*'s packet processing life cycle.

Possibly the best way to describe how *prism* (our hijack support module) works would be to trace its execution through an example:

1. In our HTTP server cluster, one of the server hosts (10.0.0.1:80) has died while

delivering a 50Mb file archive to 132.181.3.189:9000 (C). However, a new server has been started on 10.0.0.2 (which also responds to 10.0.0.1 due to IP takeover software), which will continue the task.

2. The new server calls *hijack_connection(10.0.0.1,80, 132.181.3.189,9000, 10.0.0.1 <the local machine address − used when this is a router>, return_socket, proc_filehandle, 0 <flags>, 0,0 <probe for sequence numbers>)*, accessing the *prism* API.

3. *hijack_connection* opens two arbitrary local ports − 7000 (passive, B), 7001 (active, A) − and registers a setup request with the *prism* module. It then forks off a child process that attempts to connect from the active port to 132.181.3.189:9000, sleeps five seconds, and terminates.

4. As part of the connection attempt, the kernel generates a standard TCP handshake. Every packet in this handshake passes through the *prism* module, processed by the *prism_hook* function. To simplify reading, connection endpoints will be referred to by their port numbers in the points that follow.

   4.1. On every packet received, *prism* checks that it is TCP, checks that it matches a registered port set, and decodes the source and destination. If this packet does not belong to any handled connection, or has already been processed by *prism*, it is simple passed on. Otherwise, *prism* processes the TCP sequence numbers, strips off all IP options except Maximum Segment Size (IP options are usually not critical and complicate translation)[44], recalculates the packet checksums, regenerates the packet routing (since destination addresses may change), and returns the packet to the network stack.

   4.2. SYN 7001−>7000: *prism* extracts the TCP sequence number (the first data byte sequence number for port 7001, minus one), and rewrites the packet to SYN 9000−>7000. (Step 1. in Figure 11.)

   4.3. SYN/ACK 7000−>9000: *prism* extracts the TCP sequence number, and rewrites the packet to SYN/ACK 7000−>7001 (the answer to the previous packet).[45] (Step 2. in Figure 11.)

   4.4. ACK 7001−>7000: *prism* records that the handshake is complete, and

44 Recreating the IP and TCP option settings of a previous connection would require tracking the full connection or extracting equivalent information from the original network stack. Correctly handling these options, including support for new options, is a problem for further research.

45 This does not cover the case of a TCP simultaneous open. However, since the connection will be opened by our own processes, which will not attempt a simultaneous open, this is never a problem.

rewrites the packet to ACK 9000−>7000. At this point, port 7000 believes that it is connected to 132.181.3.189:9000, port 7001 believes that it is connected to 7000, and *prism* has a record of the sequence numbers expected by both. (Step 3. in Figure 11.)

4.5. Any further packets from 7001 will not be rewritten by *prism*, and will simply elicit RST responses, since port 7000 does not believe itself to be connected to 7001. Therefore, any further packets (including FIN packets) from 7001 will close down that socket.

5. As the *hijack_connection* has now successfully received a connection request, it returns the socket for port 7000 to the HTTP server.

6. The HTTP server now wishes to resume sending data to the remote client. It does so by writing to the returned socket, which is translated into a series of packets by the operating system.

6.1. At this point, *prism* does not yet have any information on the remote sequence numbers, but has exact details on the sequence numbers for port 7000. In order to facilitate testing, it will also discard any packets from port 80 to 9000 − this connection has been hijacked. Since port 80 represents the dead client, such packets should not happen.

6.2. PSH 7000−>9000: *prism* receives data packets produced by the kernel, but cannot rewrite the sequence numbers. It simply sends forwards these onward − they will be rejected by the remote host. (Step 4. in Figure 11.)

6.3. ACK 9000−>7000: in response to the previous packet, the remote host replies with an acknowledgement containing its own sequence numbers. *prism* extracts, calculates the corrections required to map sequence numbers between ports 7000 and 9000, remaps the packet and forwards it to the server. (Step 5. in Figure 11.)

6.4. PSH 7000−>9000: at some later point the server retransmits the first packet, used by *prism* as a probe, which, along with all other packets in the connection, is mapped to matching sequence numbers for the destination.

7. At this point, bi−directional communication between the remote client and the local server is possible. This continues until either endpoint terminates the connection; on observing two FIN packets, or an RST, *prism* invalidates the mapping rule. This ensures minimal overhead due to expired mappings.

9. Load−balancing

Network load−balancing is the art of sharing a load between similar resources to improve utilisation, scalability and failure tolerance. Two varieties are typical: link balancing and server balancing. Link balancing, a basic function of many routers, lies beyond the scope of this chapter. In this section we will cover techniques used to distribute the load between network devices or servers.

Many factors influence the need for load−balancing solutions.

- The network bandwidth available to users of all levels is steadily increasing, outgrowing the ability of network servers to utilise fully.
- Protocols, in particular cryptographic applications, are becoming more complex and processing−intensive. Servers are often incapable of handling large volumes of this type of traffic.
- Replacing devices with more powerful versions can be expensive, and does not utilise the existing investment. Load−balancing offers mechanisms for making use of low−cost or outdated systems to support high demands.
- Network services are increasingly vital to many organisations. Load−balancing solutions offer improved fail−over capabilities and utilisation in comparison to simple standby schemes, improving the survivability of services.
- Manually partitioning services is administratively complex (or, in many cases, infeasible). Using load−balancing, a single system can be used to administer load sharing between servers.[46]

9.1. Approaches to load−balancing

Current server load−balancing techniques can be broken down into two classes: probe based and distribution point based load−balancing systems.

Probe based load−balancing (PLB) systems manipulate the mechanism used to contact servers during requests. Normally, server selection is based on a simple unique mapping of request characteristics to the network location. Probe based systems use an additional request resolution step, where one of a number of potential server mappings is chosen.

To illustrate, HTTP requests usually depend on a simple mapping of URL to IP address, using DNS. Round robin DNS (RRDNS) varies the result of hostname

46 In the remainder of this chapter, we will refer to a group of hosts cooperatively providing a service as a cluster, and to the individual hosts or servers as nodes.

mappings to share the load between different IP addresses.

The effectiveness of this approach to load−balancing depends on the number and nature of requests that will re−use the results of a previous mapping. This is in turn affected by performance optimisations present in the mapping process, such as intermediary caches (common for DNS), and mapping reuse conditions (a time or number of requests for which a mapping can be reused). Conversely, additional mapping probes improve the fairness of load sharing, while increasing the latency of requests − leading to a trade−off between load−balancing and request speed.

Finally, this approach requires a request resolution step. Protocols where no such step is implicit (such as HTTP requests using the correct IP address for DNS) must be modified to be balanced using this approach. Therefore, this approach is often not feasible for legacy protocols.

Distribution point load−balancing (DLB) systems make use of a virtual server to which all requests are directed. A load−balancing gateway receives requests and delegates these to real servers. Which real server is selected depends on the load−balancing device's scheduling algorithms, discussed in Section 9.3.2. The method used to transfer requests from the balancing gateway to back−end servers varies − a number of approaches are discussed in Section 9.3.1.

Distribution point systems are usually transparent to clients (and often to servers, especially at the application layer). This allows arbitrary protocols to be supported, at a slight cost in packet latency (of the order of one additional routing hop). The load−balancing gateway, however, suffers from many of the same concerns applicable to the use of a single large server − scalability, expense, and single points of failure.

9.2. Probe Based Systems

9.2.1. Round Robin DNS

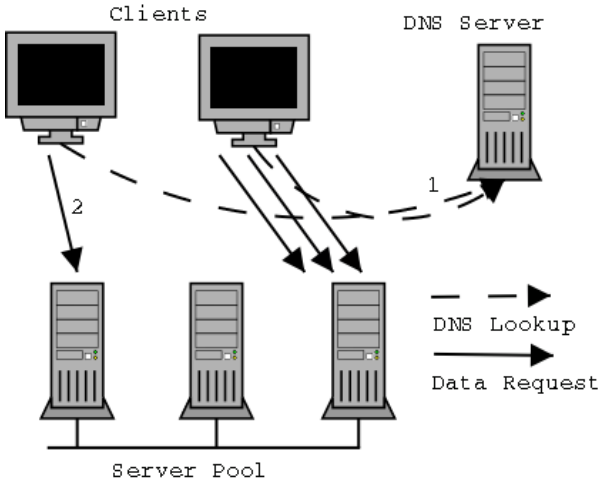

Figure 12. Round Robin DNS request structure

The Domain Name Service (DNS) [DNS] (a.k.a. DNS aliasing) is a distributed hierarchical database and protocol used to translate symbolic host names into IP addresses. When attempting to contact a remote host by name, a client first requests an IP mapping from a local DNS server. This server may, in turn, request a mapping from higher−level DNS servers, which may contact a higher or lower level server, and so on − until the name is resolved or an authorative failure is returned. Because this can be a time−consuming process (at least one packet trip per server queried), DNS servers typically cache responses to optimise future requests.

Round Robin DNS (RRDNS) [RRDNS] is an extension of this scheme where the authorative DNS server for a symbolic name holds a number of different possible mappings (refer to Figure 12). The specific mapping returned is traditionally selected in a weighed round−robin fashion (where higher capacity mappings occur more frequently in the round−robin cycle), or based on more complex criteria such as server load or network proximity.

One significant advantage of this approach, shared by many other PLB systems, is the possibility for real servers to be dispersed geographically or across networks. Server selection can make use of client location, allowing clients to use short network links directly. [Walrus] presents an implementation of a similar concept.

RRDNS is, in comparison, extremely easy and cheap to implement. It makes use of existing protocol behaviour, requires no client or server modification, and is well

supported by DNS servers. It is inherently supported by most current IP protocols, requiring no additional request latency.

RRDNS is, however, not a perfect solution. Caching in DNS servers and end clients causes multiple requests to use the same mapping, reducing the granularity in load−balancing. As a result, multiple requests from the same clients, and other clients sharing part of the DNS path, will be directed at a single real server [NCSA]. Combating this is possible by forcing DNS responses to expire rapidly (by lowering DNS response TTL values). However, this increases request latencies and is ignored by many clients.

DNS requests also carry no information on the request requiring address resolution. Therefore, RRDNS systems cannot distribute mappings based on request complexity or content. RRDNS is incapable of differentiating, and hence balancing, lightweight and computationally intensive requests to the same virtual server.

Finally, RRDNS has no inherent support for detecting or recovering from node failures. If a server node crashes, a conventional DNS server will continue to direct traffic to the unresponsive address. Even if a modified RRDNS server which tracks server responsiveness is used, there is no mechanism in DNS to invalidate existing address mappings held on clients or DNS server caches.[47]

47 Mechanisms to recover from this situation have been developed using IP takeover and heartbeat systems [LHA], but are incapable of distributing the failed node's workload, and only works for nodes that reside on the same subnet.

9.2.2. Berkeley Smart Clients

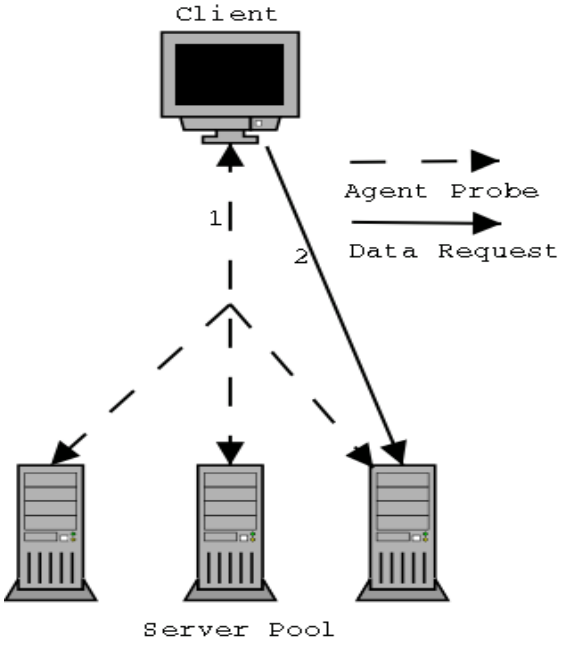

Figure 13. Smart Client based load−balancing using agent−based monitors

The Berkeley Smart Client system [SmCli] is an attempt to shift load−balancing from the server side to clients. For every service, each client runs a director Java applet that maintains lists of candidate servers and scheduling information. This applet state is maintained by continuous communication with servers, using probing and lazy[48] updates from servers (Figure 13).

When a client wishes to use a service, a modified Java−based Web browser contacts the director applet. This applet selects a server to contact, based on server profiles or some heuristic mechanism.

The Smart Client design reduces the problem of increased request latency with independent state maintenance, at the cost of continuous network and processing overhead. In addition, this technique relies on the use of modified clients (in particular, relying on the extensibility of Web browsers using Java), and extended server capabilities. The director applet may be able to leverage service−specific features (such as FTP server mirroring), but would require service−specific directors.

9.2.3. HTTP Redirect

The SWEB system [SWEB] uses HTTP protocol features to distribute load among servers. When an HTTP request is received, the SWEB redirection server evaluates

48 By allowing updates to accumulate before sending, the number of messages and overhead associated with maintaining the applet can be reduced.

which node is likely to offer the best response time (based on the cost of redirection, node loads, and the nature of the request). If the selected node is not the evaluating node, an HTTP redirect response is generated, informing the client that a different node should be contacted (Figure 14).

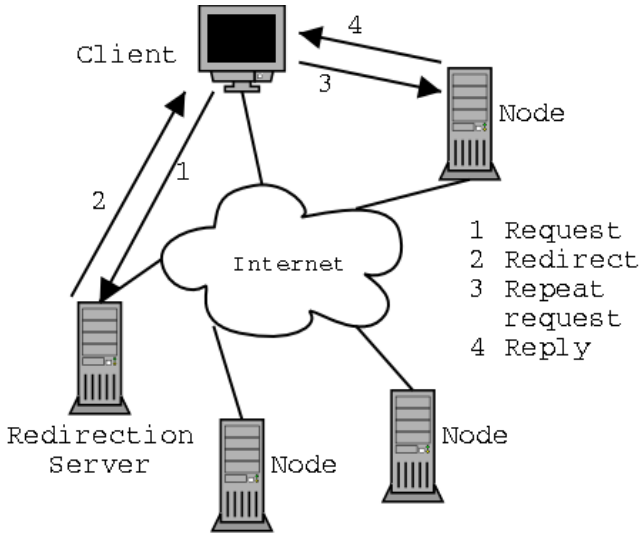

Figure 14. Load balancing using a redirection server

This mechanism offers good scalability to HTTP−based services, can allow geographically dispersed clients, and has no low−level networking or topology requirements. It does require modifications to the HTTP (or equivalent) server system, and does not apply to protocols without a redirection mechanism.

9.3. Distribution Point Systems

Distribution point load−balancing systems make use of a load−balancing gateway to split the offered load between real servers (or *nodes*). Requests are directed at a virtual server (representing the joint service offered by the cluster), with the routing path passing through the load−balancing gateway. From there, requests are forwarded on to real servers via one of the techniques described in Section 9.3.1. The exact server to which a request is submitted depends on the scheduling scheme used, as described in Section 9.3.2.

Since many of the same techniques used in distribution point schemes are utilised by the implementations covered here, we will discuss these techniques first. For more detail on these methods, refer to the implementations described in Section 9.3.3.

9.3.1. Delivery Mechanisms

Distribution point load−balancing systems must have some mechanism for

distributing requests to the appropriate handler nodes. This section briefly discusses a number of the approaches used.

9.3.1.1. Data Link Switching

In data link switching, the load−balancing gateway takes the form of a network switch connecting the server cluster and the external network. Each node is configured to respond to the virtual cluster IP address, and contains similar software configurations.

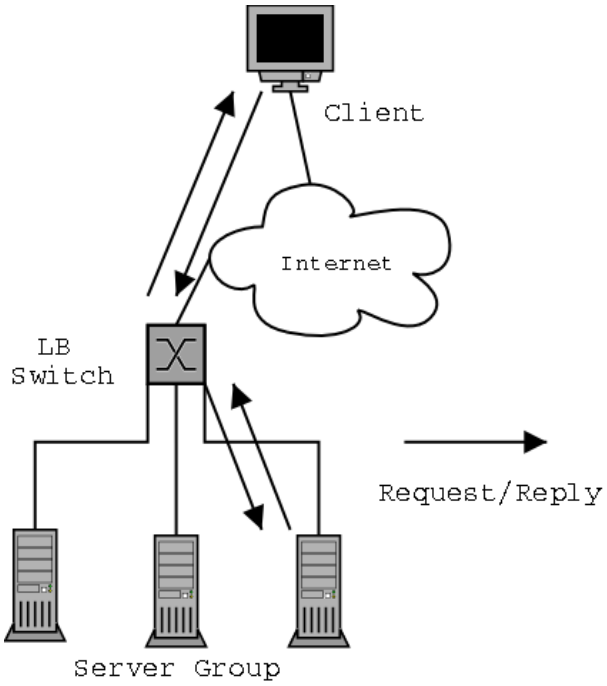

Figure 15. Load−balancing switch configuration

The load−balancing switch receives incoming packets, examines the packet payload to determine the appropriate node to forward to, and resends the packet directly to the appropriate data link MAC address of the selected node from the server group (Figure 15).[49] Replies are simply passed back to the external network link.

This technique requires specialised hardware (the load−balancing switch itself), but works with any node configuration. Since no packet modifications are required, it can be extremely fast − though the same route is used for incoming and outgoing traffic.

9.3.1.2. Direct Routing

Direct routing is a variation on data link switching, where the load−balancing gateway takes the place of an incoming gateway to the physical segment on which all nodes reside. Nodes are configured to respond to the virtual cluster IP address, but

49 It also compensates for conflicting ARP responses from nodes, maintaining MAC address tables. In effect, it falls somewhere between ISO layer 2 and 3 − with higher layer comprehension.

with ARP disabled, and contain normal software.

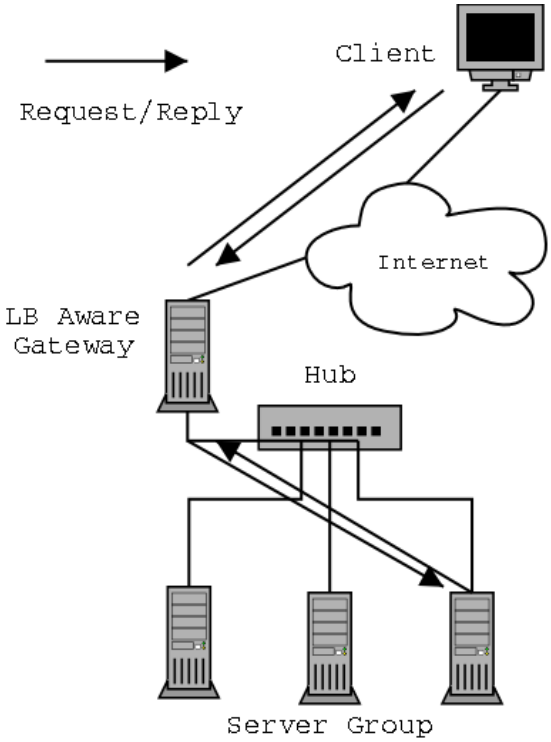

Figure 16. Load−balancing using a MAC−layer routing gateway

When the load−balancing gateway receives a request, that packet is forwarded directly to the MAC address of the node selected by the scheduling mechanism (Figure 16). Since ARP does not function for the virtual cluster address, alternative mechanisms (such as static configuration or linking servers to secondary IP addresses) must be used for MAC address resolution. Responses are returned directly to the client, possibly via an alternative gateway.

9.3.1.3. Network Address Translation

The basic concepts of NAT have already been described in Section 2.1.5. On receiving a request, a NAT router selects a real server address to handle that packet. The destination address and port numbers of that packet are then rewritten to match that of the real server, and the packet is routed onward. On reply packets the source address and port number are rewritten to match the virtual cluster details, as stored from the earlier request. [LSNAT]

The load−balancing NAT router must be located on the routing path to and from nodes (and the virtual cluster address) − such as on an exterior gateway or firewall (Figure 17). Server nodes do not require the virtual IP, and can be configured using normal IP numbers. When the NAT router is disabled, they can continue to function

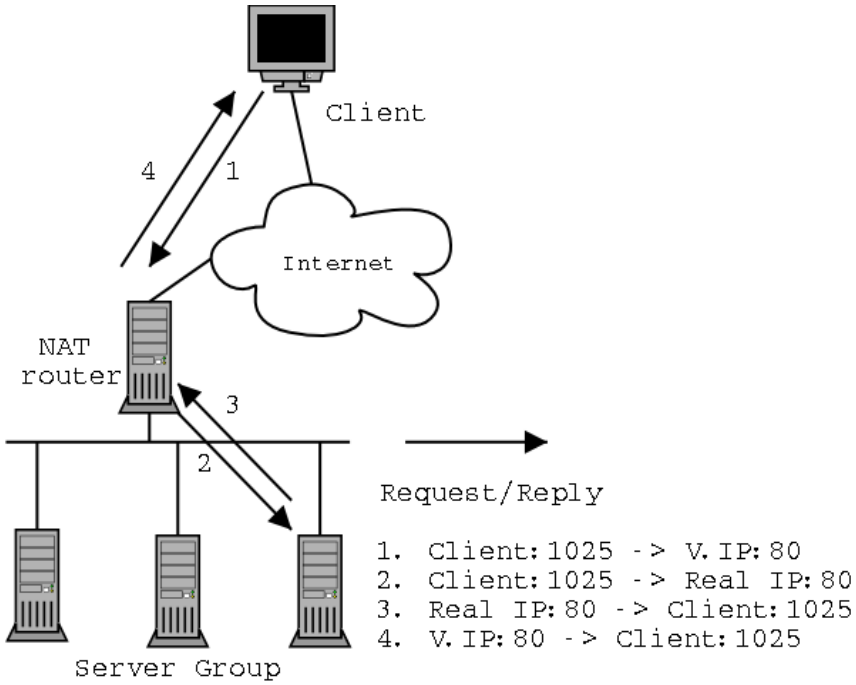

Figure 17. Load−balancing NAT gateway configuration

as normal servers using their unique IP addresses (assuming some routing path is available). It may in fact be beneficial to configure the most powerful node to have the same IP as the virtual cluster IP. Should the NAT device fail, it can be replaced with a conventional gateway or router.[50]

This mechanism requires packets from both directions to pass through the NAT router. Since IP and TCP addressing information is modified, this also requires packet checksums to be recalculated, and introduces a relatively high performance overhead. Problems noted with NAT and complex or stateless protocols in Section 2.5 also apply here.

However, this technique is a natural extension to existing NAT and firewall services. In this configuration, the additional overhead of load−balancing is minimal, and involves few additional complications.[51] In this situation, NAT−based load−balancing is a very natural match.

9.3.1.4. IP Tunnelling

In IP tunnelling, the distribution host itself is configured with the virtual cluster IP. Nodes must accept packets directed at this virtual IP (often done by aliasing the V.IP

50 By adding a second, high−cost route into the cluster network, some protection from NAT device failure can be attained. If the NAT device fails, the less−preferred direct route is used, accessing the back−end nodes directly.

51 One possible complication is maintaining the cross−connection state for applications such as session−based HTTP. Conventional NAT takes care of most of these.

to be a loopback address), but this V.IP should not be routable to nodes. Nodes are otherwise configured with conventional software.

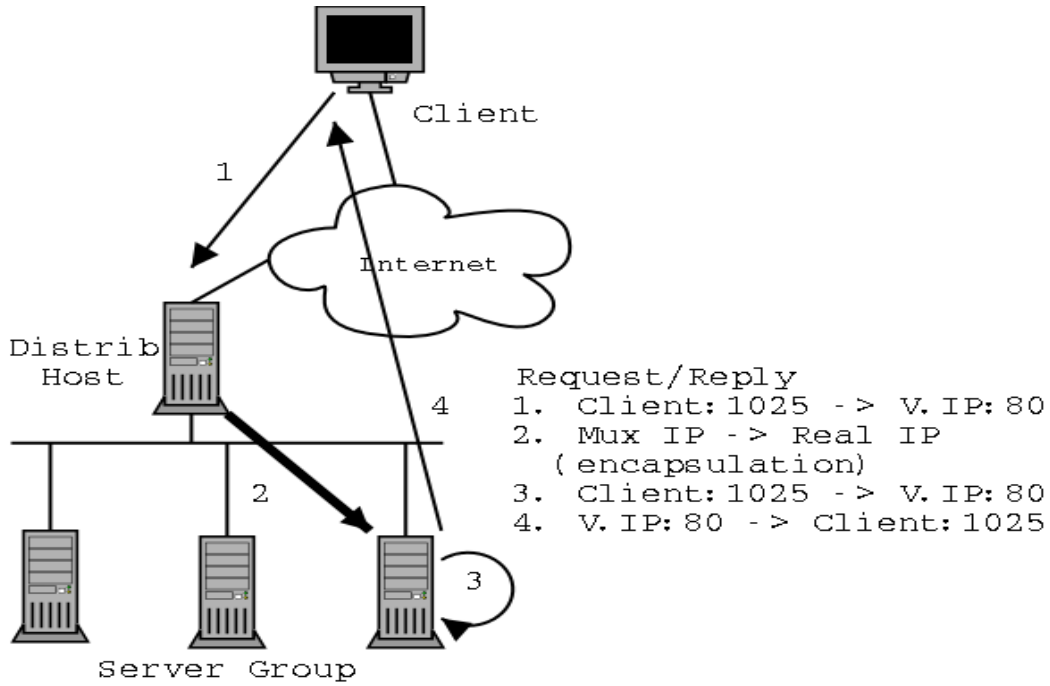

Figure 18. Load−balancing structure using IP tunnels

When a request packet reaches the distribution host, it is encapsulated in another IP datagram directed to the selected real server's unique IP address (Figure 18). On arrival, the real server unwraps the request packet, and responds as if it had been received normally. Response packets are sent directly to the client.

In many protocols (such as HTTP), the volume of server responses to requests far outweighs client request packets (and acknowledgements). With geographically dispersed nodes, it is possible to select the real server closest to the client. At this point, the performance improvement of direct responses can outweigh the cost of encapsulating and retransmitting requests significantly. Additionally, this allows independent network links to be used for different real servers.

9.3.1.5. Filtered Broadcast

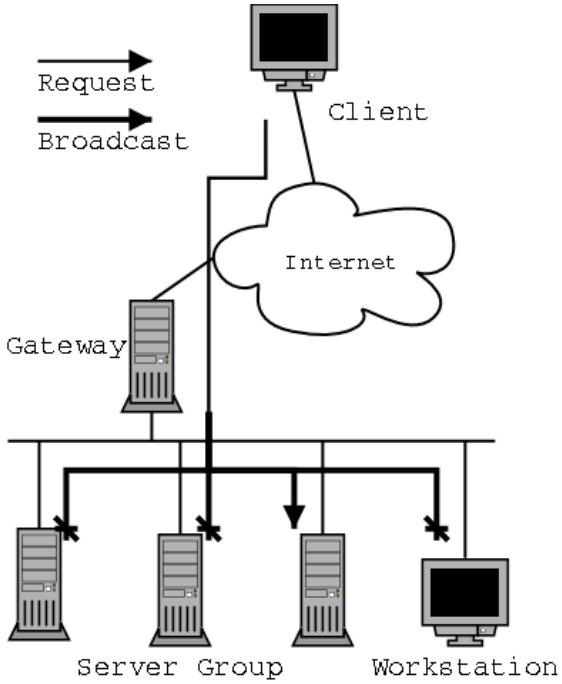

Figure 19. Filtered Broadcast request distribution

All of the preceding delivery mechanisms rely on server node selection first, then delivery. The filtered broadcast mechanism consists of statically configuring the cluster subnet gateway to associate the virtual cluster IP with a broadcast MAC address. When a request is received, it is broadcast to all nodes (Figure 19).

Each node must have two low−level operating system modifications in place. Firstly, it must accept packets sent to the broadcast MAC address as it would unicast packets directed to itself. Secondly, nodes must have a selection mechanism in place to determine whether a given packet should be discarded or handled (effectively acting as a distributed scheduler).

Unlike many of the previous mechanisms, this approach does not have a single point of failure. In addition, due to the nature of many LAN technologies, a broadcast message takes as long as a unicast message. In order to avoid unnecessary handling of broadcasts by non−cluster hosts, the cluster subnet should be separate from other devices.

9.3.1.6. Reverse Proxies

A reverse proxy acts as an agent for the client. On receiving a request, it parses the request, and contacts a real server directly to obtain the required information [RProxy]. The proxy then generates an answer to the request and sends it back to the

original client (Figure 20). Effectively, this technique is to transparent proxies as load−balancing NAT is to conventional NAT.

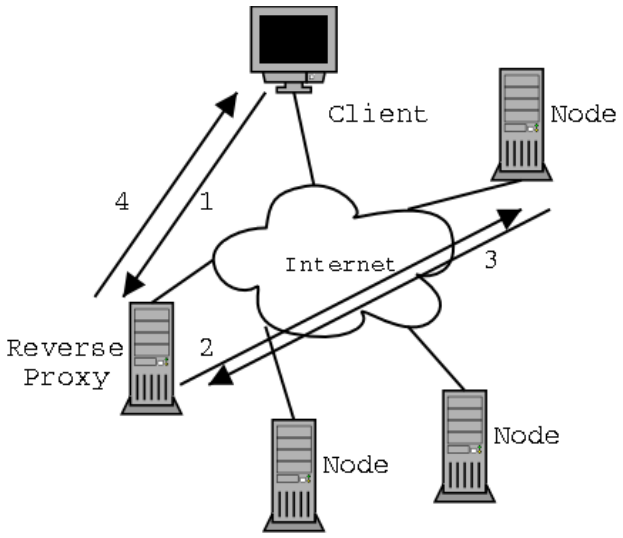

Figure 20. Load−balancing using a Reverse Proxy

This approach has many of the same problems as general proxies: protocol−specific implementation, data forwarding overhead[52], and difficulties in offering high availability. In addition, this form of load−balancing may introduce triangular routing.[53]

9.3.2. Scheduling Schemes

The key to effective load−balancing lies in selecting the best node to handle every request. The goal is for every node to have a similar load relative to the processing power of that node. Several factors make this process difficult:

- The handler for a request often has to be selected before the details of that request are available, complicating estimation of request complexity.
- The node allocated to a request can be constrained by existing conditions. Session−based dialogues constrain different requests to be allocated to the same node, further complicating load prediction.

52 [TCPOpt] presents a technique suitable for eliminating most application forwarding overheads introduced by the use of a proxy.

53 Triangular routing is where a request to a nearby resource is routed via a distant point, instead of following the short side of the triangle formed by the three devices.

- With the use of NAT, client−side proxies, tunnelling (including VPN) and similar techniques, the correlation between client identity and packet addressing information is broken down. Maintaining sessions on a per−client basis may be infeasible, reducing the flexibility of load dispersion.
- Traffic patterns follow a heterogeneous distribution across time and the packet address space. This complicates simple division schemes.
- Shifting requests or sessions between nodes is computationally expensive, if possible at all. Immediate correction of imbalances is difficult.
- Maintaining accurate node load statistics introduces additional overhead and complexity into the load−balancing system. In addition, current node processing load may not reflect the future load of sessions allocated to that node.

In this section, we describe some approaches to scheduling requests for load−balancing used in current systems.

9.3.2.1. Weighted Round Robin

Round robin scheduling consists of allocating a request to each node in turn. Weighted round robin gives each node a number of turns based on their relative processing power.

Possibly the simplest form of scheduling, this method is best known for its use in RRDNS. In distribution−based systems, it requires tracking all connections at a central point, making this method unsuitable for broadcast distribution and other distributed schemes. It also ignores request content, which can result in unbalanced loads when requests vary greatly in processing cost. Refer to [NCSA] for an evaluation of this method.

9.3.2.2. Weighted Least Connection

Least connection works by allocating a new request to the node with the least existing requests. Weighted least connection measures the number of requests a node has relative to its capacity.

Implicit in this method is the assumption that the processing load of a request is linearly dependent on the time taken for that request. Since network bandwidth can constrain the time taken to complete a request, this assumption does not always hold true. As with round robin, least connection scheduling requires request tracking, preventing its use in distributed schemes.

Intuitively, this approach should produce well balanced nodes. In practice, this

method suffers when used to distribute highly variable length requests, generating comparable results to round robin.

9.3.2.3. Address Mapping Hashes

A request is considered to be uniquely identified by the combination of local and remote packet addresses − effectively, by the tuple (local IP, local port, remote IP, remote port). An address mapping hash is a calculation used to translate this tuple into a single integer representing the node number for that request.

For example, NAT based distribution modifies the local port address of a packet. A typical simple hash suitable for use in such a system is (from [Netfilter]):

Node number = (local IP + remote IP + remote port) mod (Number of nodes)

This method is very fast, and requires no state tracking of request allocations. However, it has no concept of session, and is vulnerable to addressing concentrations in the request stream. In addition, changes to the number of active nodes may only be possible by disrupting current connections.

9.3.2.4. Minimal Node Load

In the minimal node load approach, the scheduling device maintains load statistics for every node. A new request is assigned to the node with the lowest current workload, based on some internal benchmark − CPU idle time, free memory, ping time, or some compound measure.

This method not only requires request tracking, but also requires tracking of node conditions − server CPU utilisation, memory allocations, etc. Nodes may need to be modified to facilitate extraction of a useful performance measure. Since node failure detection requires some node monitoring, the additional overhead of performance monitoring may be minimal.

9.3.2.5. Response Time Prediction

In proxy and redirection systems, the load−balancing host extracts and interprets request structures. This allows these systems to pick the node expected to offer minimal response time − based on redirection costs, node load, topological location and request characteristics.

Since this method can accurately predict the cost of requests, it offers very good balancing. The processing load involved on the load−balancing device may be significant, however, and the caching of request details changes the end−to−end

characteristics of requests (breaking transparency). This method also requires protocol−specific (and possibly application−specific) understanding of requests, making it unsuitable for general use.

9.3.2.6. Node Affinity

A simple heuristic for identifying sessions running across multiple requests is that the requesting IP address is constant across requests. By guaranteeing that all requests from a specific address will be directed to the same node, a load−balancing scheduler can support sessions without protocol−specific knowledge, and can support many complex protocols that use multiple connections (such as FTP). This association of node and remote address is termed affinity, or sticky connections.

The discussion above uses request and connection interchangeably. This simplification holds for some IP−based protocols, notably HTTP, SMTP and others. More complex protocols, such as RPC−based services, FTP and session HTTP use multiple connections − for these services, a scheduler should group associated connections.

Address mapping hash functions, which translate packet addresses into a node number, are a particular problem. Due to the presence of multiple connections, port numbers cannot be relied on, and the packet destination would always be the cluster virtual IP. Therefore, the scheduling hash would have to be calculated based solely on the remote address − further reducing the granularity of load division.

As with NAT, ICMP error messages must be handled based on the carried IP and protocol headers, rather than the outer packet header.

9.3.3. Distribution System Implementations

9.3.3.1. Linux Virtual Server

The Linux Virtual Server (LVS) [LVS] system is a combination of open−source network and application software that supports load−balancing and high availability on Linux systems. Load distribution is implemented via direct routing, NAT, or IP tunnelling, using (weighted) round robin or least connection scheduling (as described in Sections 9.3.1 and 9.3.2).

Supported features include warm standby backup devices via *fake* [fake] and *mon* [MON], node affinity and FTP support, and correct ICMP forwarding. Node failure handling is managed by removing failed nodes from the scheduling algorithm.

9.3.3.2. Cisco LocalDirector

The Cisco LocalDirector [LocalDir] is one of a family of Cisco load−balancing systems. It offers data link switching of requests, with extensions to the Cisco router IOS software offering NAT facilities. Scheduling uses round robin, least connection or load−based selection (using Cisco's Dynamic Feedback Protocol [DFP]).

The Cisco solutions also offer a number of advanced features, including:

- TCP handshake monitoring to detect server overload or failure.
- Support for a hot standby backup Cisco LocalDirector, automatically maintaining synchronised internal states.
- Automatic handling of server failures. On server failure, nodes are removed from scheduling, with support for last−resort HTTP redirection. It automatically recognises nodes that return to service.
- Support for a variety of affinity mechanisms, including simple affinity, SSL session ID tracking, HTTP sticky cookies, and support for common complex protocols.
- Connection cap (maximum connection) and slow start mechanisms to protect servers from overload.
- Support for distributed load−balancing using Cisco's Multinode Load−Balancing (MNLB) [MNLB]. On receipt of a request, a Cisco router can contact a LocalDirector for node scheduling, handling delivery directly. This allows greater scalability at the cost of some request setup latency.

9.3.3.3. ONE−IP

The ONE−IP [ONE−IP] system offers two delivery mechanisms: filtered broadcast and direct routing. Scheduling is done via a simple hash (where *node = remote IP mod number of nodes*) − implicitly granting connection affinity. When a node fails, this hash is modified slightly: if the node selected is down, the hash is recalculated based on *(n−1)* nodes and mapped to the remaining cluster. This allows nodes to be removed without disrupting existing requests. Adding nodes can only be done by taking the cluster offline.

9.3.3.4. Other Examples

Other load−balancing systems[54] include Berkeley's MagicRouter [MR] and the Linux Netfilter BALANCE target [Netfilter], both using NAT−based delivery. Scheduling

54 Refer to [Prices] for details and prices of other commercial examples.

uses round robin, random, hashed or load−based algorithms.

The IBM Network Dispatcher [NetDisp] uses a direct routing mechanism and schedules according to existing server loads. In addition, it supports a variety of advanced features, including affinity, request forwarding to other clusters, and high availability features.

10. High Availability

High availability covers a range of techniques used to minimize the effect of network device failures. In the ideal case, a highly available system has sufficient redundancy built−in so that, when a device fails, the workload of that device is taken over by other devices. This takeover should be completely transparent to clients − maintaining existing configurations, state tables and buffers.

In practice, it is possible to distinguish three forms of redundancy:

1. **Cold standby** − A preconfigured backup device is available, but powered down. Fail−over consists of manually removing the failed device, replacing it with the backup, and booting the backup (typically taking some minutes at least). This loses all transient device state, usually disrupting any services in use.
2. **Warm standby** −A preconfigured backup device is powered up and connected to the network, but does not track the primary device's state. Fail−over may be done via software (such as *fake* [fake], allowing fail−over to take effect in seconds), but disrupts any stateful services in use.
3. **Hot standby** − A preconfigured backup devices is running alongside the primary, tracking that device's internal state. Fail−over is done via software, and may retain some of all internal state − possibly avoiding any service disruption.

A number of commercial and research high availability systems have been developed in recent years. In the remainder of this section, we will discuss a number of these.[55]

10.1. Cisco Hot Standby Router Protocol

The Cisco Hot Standby Router Protocol (HSRP) [HSRP] outlines processes for controlling which of a group of routers should act as a gateway for a network. It consists of a group of routers that share a common network segment, virtual IP and MAC addresses, and HSRP group number. From this HSRP group, the highest priority router is elected as the active router − which accepts and forwards client packets − and one as a standby router. The active router periodically transmits HELLO messages with a TTL value, which are tracked by other group members. When the TTL expires (or the active router resigns), the standby router becomes active (sending a gratuitous ARP message to retrain switches), and some other router becomes standby router.

55 Many of the high availability techniques described here are proprietary, and are discussed based on publicly available product literature.

In common with our packet delivery mechanism (see Section 11), every router in the HSRP group shares a virtual MAC address (preferably in addition to their conventional MAC address). Only one router actually receives client packets, however.

HSRP offers a mechanism for controlling which of a group of routers is used by clients. It does not offer any state synchronisation between routers, and hence does not support routing based on interactions between clients and the active router. This places it into the warm standby form of redundancy.

10.2. Watchguard Firebox High Availability

The Watchguard Firebox high availability option [WGHA] is a good example of a commercial firewall with fail−over capabilities. It consists of primary and backup Fireboxes, both connected to the same network segments (see Figure 21). The backup Firebox maintains a heartbeat connection to the primary unit − when the heartbeat is lost, the backup takes over.

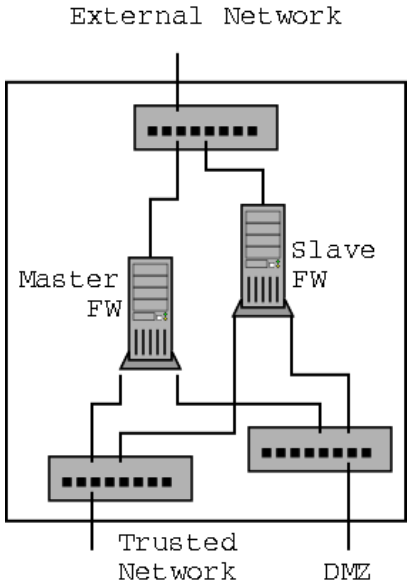

Figure 21. High Availability Master/Slave network topology

The Watchguard Firebox is a hybrid stateful filter and proxy firewall. On fail−over, however, only the filtering state is maintained across Fireboxes − proxied connections are lost.

10.3. Checkpoint Firewall−1 High Availability Module

The Checkpoint Firewall−1 high availability module is an extension to the Checkpoint Firewall−1 software that maintains stateful packet filtering and VPN state between primary and backup firewall hosts [CHA]. When the primary firewall fails, the highest priority backup node takes over.

The Checkpoint high availability module effectively synchronises internal states between firewalls in a cluster. While supporting host standby high availability, this also allows the use of load−balancing systems such as Stonebeat [Stonebeat] or load−balancing switches.

This approach has two notable limitations, however. Firstly, the Firewall−1 has no internal support for proxies or content filtering.[56] These functions are therefore not included in the fail−over configuration. Secondly, the stateful filter table is synchronised between all firewalls in a cluster. This prevents the system from being arbitrarily scalable − the state table is subject to the same limits as in a single−node system.

10.4. Linux High Availability Project

The Linux High Availability Project [LHA] is an open−source implementation of a warm standby system. Essentially, a server can be configured to monitor another host via the *heartbeat* component. When a server becomes unresponsive, the monitor can take over the duties of that server using *fake* − configuring the IP locally and using gratuitous ARP to redirect clients.

This system, used in conjunction with the Linux Virtual Server project (see Section 9.3.3), offers a load−balanced, warm standby system with network−level fail−over.

10.5. Conclusion

A number of high availability offerings are available, ranging from simple cold standby to hot standby with load−balanced clusters. None of the currently available systems, however, support the full range of firewall techniques in a scalable, load balanced, hot standby configuration.

In this thesis we outline an approach offering all of these functions − refer to Section 4 for more detail.

56 Using the Content Vectoring Protocol [OPSEC], these functions can be passed over to external applications running as part of the firewall.

11. Distributing packets to cluster nodes

In many load−balancing systems, packet distribution is handled by a load−balancing gateway, which dispatches requests to nodes using direct connections (similar to a switch), or by readdressing (or encapsulating) packets (similar to a NAT gateway). In probe−based systems, the client differentiates nodes by address, and request distribution happens through the normal routing process.

The gateway approach has problems of scalability, and topological constraints − this approach does not lend itself to serving clustered gateways. One approach used in the ONE−IP system is to equate the virtual server IP address with a data link broadcast address − effectively broadcasting incoming requests. This approach, while avoiding the problems of a load−balancing gateway, does not lend itself to switched environments − since all requests would be broadcast to non−cluster hosts as well − and requires modifications to the host operating system.

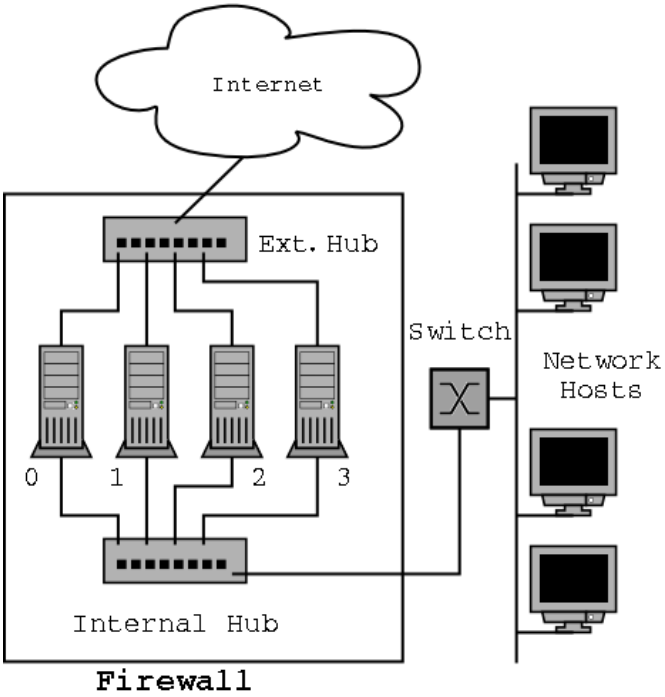

Figure 22. Load−balancing firewall structure using a shared MAC address

We have developed a simple system[57] for distributing data between nodes, based on the use of shared MAC addresses between nodes. This allows the use of switched networks between cluster and hosts (Figure 22), and allows nodes to be built using

57 This approach is similar to the Cisco HSRP [HSRP] protocol in the use of a shared MAC address between nodes.

standard software.

11.1. Network Topology

Figure 23 outlines the configuration of our testbed network implementation[58]. The IP and MAC addresses of hosts are as shown, with a number of details of particular significance:

1. Nodes 1 and 2 (see Figure 23) each have two network interfaces, allowing them to be used as gateways between the internal and external networks.
2. The MAC addresses and one of the IP addresses of both nodes are identical on the internal and external networks. ARP requests for the shared IP address will always result in the same MAC address. RARP requests on the MAC address will resolve unpredictably, depending on which reply is accepted.
3. Each node maintains a unique IP address per interface (e.g. 10.0.1.1). This address is used for communications involving specific nodes, such as GLOB (see Section 12) unicast messages and general protocols (Telnet, SSH).
4. The nodes must be located on a shared link (in this case a hub), to ensure that packets are received on all interfaces. Other devices on the same network may be connected to this shared bus via a switch − since there is a single MAC address between nodes. This allows the configuration shown in Figure 22, avoiding the need for additional routers in a small network.

58 This network was implemented using Linux kernel 2.2 and 2.4 based nodes, using Intel EtherExpress Pro network cards and *ipchains* or Netfilter−based packet filtering.

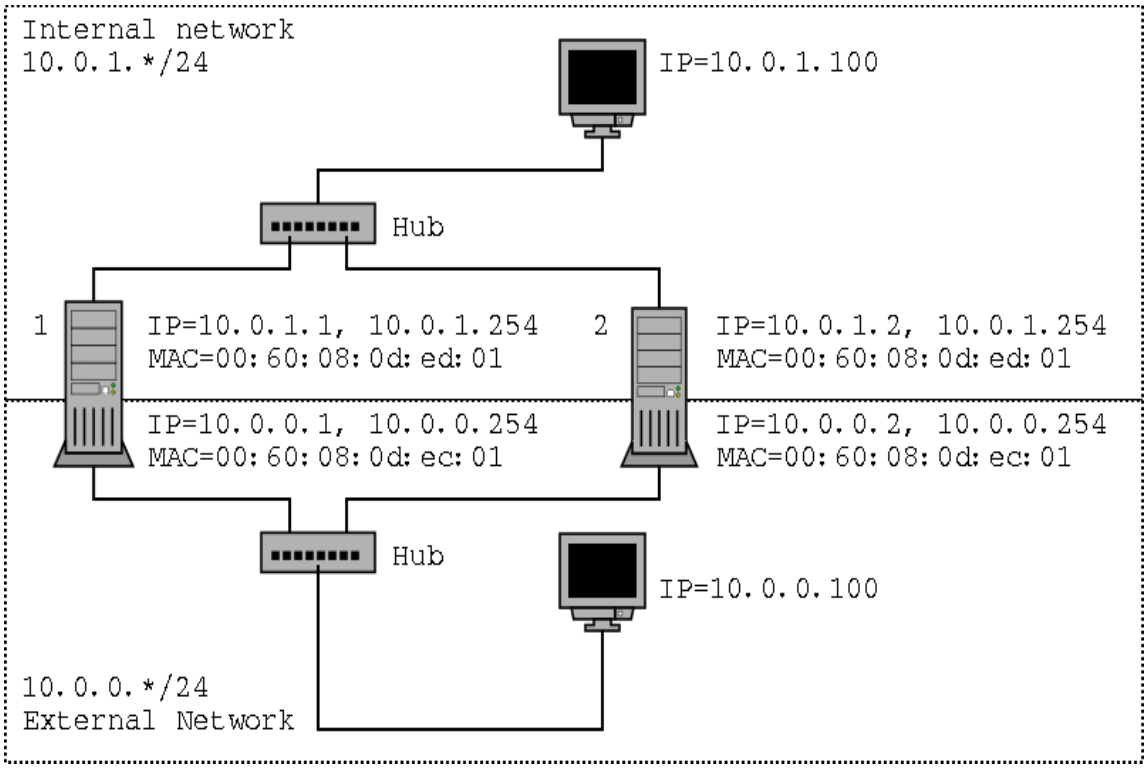

Figure 23. Testbed distribution network configuration

5. The current design requires nodes to have identical performance characteristics. Different configurations would be possible, but would require more complex load distribution software.

11.2. Node Configuration

The steps involved in building such a configuration (Figure 23) are:

1. For each network (internal, external, DMZ, etc.) that the cluster will be attached to, ensure that a shared link (generally a network hub) is connected to the appropriate node interfaces.
2. For each network, allocate a shared virtual IP address (e.g. 10.0.1.254), and a shared MAC address (e.g. 00:60:08:0d:ed:01). Also, allocate unique IP addresses (e.g. 10.0.1.1) for all nodes per network.
3. Set the MAC address for interfaces connected to a network on each node to the shared MAC address defined for that network.
4. Set the IP address of each node interface to the unique IP address allocated for that node and network.
5. Finally, add the virtual IP address for each network to all node interfaces connected to that network.

At this point, the nodes are capable of unicast communication via their unique IP addresses, and can act as a redundant gateway system. Experimentation on the internal client (10.0.1.100) gives the following results:

1. Pinging a unique node IP address results in multiple responses. The IP stacks on these hosts do not appear to check destination addresses, and all nodes respond.
2. Pinging the local or remote virtual IP address (10.0.0.254, 10.0.1.254) has an identical effect (multiple responses). In this case, this is correct, in contrast to the previous observation.
3. Pinging the remote client (10.0.0.100) through the gateway cluster produces $n * n$ responses (where n is the number of nodes). Each node receives a ping request and forwards it − resulting in n identical requests, and n responses at the remote host. Each of these responses are similarly duplicated.
4. Disconnecting up to $n-1$ nodes while pinging produces no disruption of service.
5. Attempting to connect to the virtual server IP via TCP succeeds, up to the first client data packets. During the initial handshake, one arbitrary node's SYN/ACK is accepted, and that determines sequence numbers for the connection. On receiving client data, all other nodes respond with an RST command, since the data does not match their sequence numbers − terminating the connection.
6. Testing a ping flood through a single−node cluster produces no packet loss. Attempting the same test on a two−node cluster results in dropped packets, as a result of the duplication effect noted before. By amplifying the incoming flood of packets, the gateway exceeds its own capacity.

Therefore, the testbed can be described as a fully redundant configuration with no failure recovery latency. However, two problems remain: packet duplication (Observation 3.) and node/client synchronisation problems (Observation 5.). These problems can be trivially avoided using a packet filter on each node, limiting the traffic accepted to be disjunct from what all other nodes accept. For example, node i handles traffic with an external address (*modulo* n) of i. Warm standby fail−over can be handled by a simple heartbeat mechanism.

Alternatively, the Generic LOad−Balancing (GLOB) system described in Section 12 attempts to offer more balanced loading and less disruptive fail−over handling.

12. Generic Load−Balancing[59]

In Section 9, we discussed a number of currently available load−balancing techniques. When applying these techniques to network gateways (see Figure 22), however, the requirement that all traffic associated with a request must pass through the same node presents some problems.

Probe based systems rely on the client to nominate a service node[60]. In order to support load−balanced gateways, this would require the client to specify the exact routing path taken by request packets[61] − breaking IP routing's link failure recovery. In addition, this would greatly increase the complexity of clients. Return packets present another problem − the gateway would have to ensure that return packets pass through the same node − by techniques such as NAT or conventional proxies. In addition to limiting the filtering options on firewall gateways, this also breaks transparency.

Distribution point load−balancing systems would seem to be a better match, because of the localised scope of such systems. Unless some technique is used to constrain replies into returning to the same gateway node, there would need to be a load−balancing device on every network link to which the gateway's cluster is connected. [FWLB] demonstrates the implementation of such a system, where outgoing requests are balanced between proxy firewalls. An alternative to proxies would be the use of port−pool NAT, guaranteeing that responses are returned to the firewall gateway's unique address.

In order to load balance all traffic directed into a clustered gateway while allowing the use of packet filtering or transparent proxies, a distributed load−balancing gateway would be required. Such a system would have to compensate for NAT effects on the gateways, and correctly channel related packets into the same node on all interfaces. In addition, such a system would have to maintain internal synchronisation between load−balancing devices, without forming a new performance bottleneck or single point of failure. To the best of our knowledge, no such system currently exists.

---

59 A paper based on this chapter has been accepted to the 9th IEEE International Conference on Networks, and will be presented in Bangkok, October 2001.

60 In this discussion, a distributed server, or cluster, is a collection of individual devices, or nodes, connected by network links.

61 By modifying the client's default gateway for first−hop gateways, or using IP routing options. Note that many systems ignore or block packets with IP routing options imbedded.

In this chapter, we will present a solution to this problem. Essentially, it avoids the issue of distributed load−balancing devices by placing that functionality into the nodes themselves. This Generic LOad−Balancing system (GLOB), while applicable to any distribution point load−balancing problem, handles the problems of multiple network connections and routing traffic effortlessly.

12.1. Overview

The basic concept of the GLOB system lies in letting every node in the gateway cluster pick out the traffic for which it is responsible according to negotiated shares. The technique described in Section 11 is used to deliver all packets directed at the firewall cluster to every node. Nodes are responsible for discarding packets they are not responsible for − where the traffic shares of nodes are negotiated in a peer−to−peer fashion.

This system is based on a number of concepts and expectations:

- All possible connections can be mapped to an internal and an external port[62]. By dividing this port space into adjoining disjunct sections, it is possible to generate a unique mapping of a connection−to−port subdivision (referred to here as *rule*, essentially a static packet filtering rule where the rule outcome is a node number).
- Localised load−balancing is only suitable for applications where processing power is in more demand than network bandwidth. In addition, external (WAN) network links have less bandwidth than intra−cluster links. The system requires server rather than link balancing, so a distribution system makes sense.
- Stateless packet filtering of the network load expected by the system as a whole does not significantly affect the performance of nodes, while stateful packet processing does present a nontrivial load. The nodes can handle the raw data rate, but not the complex processing required to track the traffic content.
- Protocols that require multiple connections (such as FTP, and session−based HTTP) do not require completely arbitrary ports. In particular, such services can be distinguished from single−connection protocols, and do not span multiple IP endpoints (multicasting protocols require similar handling with address−based splits)[63]. For GLOB to work, it must be possible to break down the address space

62 This holds for TCP, UDP and related ICMP traffic. For other forms of traffic, address−based rules (an extension to this prototype) can offer load division.

63 For example, FTP is a complex protocol, but all outgoing connections to FTP servers will match local ports (1024−65535) and remote ports (20,21,1024−65535) − effectively excluding all HTTP connections. In other words, it must be possible to create some form of port division.

into smaller sections, without fragmenting complex services.

Under GLOB, every gateway node receives all traffic directed at the cluster, where these packets are passed through a packet filter before further processing. This packet filter consists of rules representing sections of the packet port space allocated to this node. If a packet matches any rule, it is passed on to the remainder of the networking system for local processing − otherwise it is discarded, under the assumption that some other node will accept its copy of the same packet.

Every node also monitors its accepted load relative to the total load. If a node is overloaded, it sheds part of its rule set to a different, less loaded node. This mechanism ensures a balanced load between nodes, and acts as an automatic slow−start mechanism for new nodes. Should a node fail, its rules are taken over by other nodes, maintaining service.

Every rule represents a segment of the range of possible packet port addresses, adjoining but disjunct from all other rules. Over time, a rule also represents a processing load, based on the number of packets handled that matches that rule, and therefore represents a collection of protocol connections. A rule is associated with two nodes: the primary node (which considers it to be a local rule), which actually handles associated packets, and a backup node, which will take over that rule if the primary node fails or becomes overloaded. The backup node is also responsible for maintaining sufficient state information to support fail−over of higher−level functions for connections using that rule.

The GLOB system described here has been developed on a Linux kernel version 2.4 based system, using the Netfilter [Netfilter] framework. However, any architecture capable of modifying a network interface's MAC addresses and supporting modifiable packet filtering structures with sufficient reporting[64] could be used.

12.2. Modelling GLOB

The basic implementation of GLOB consists of a combination of packet filtering kernel capabilities and a userspace controller (*lbmodel*). Due to time constraints and implementation issues, some of which are covered later in this section, we evaluated the viability of the GLOB scheme in a model framework in comparison to other scheduling schemes in use in mainstream load−balancing systems.

64 GLOB needs per−rule traffic counters, and the ability to change the packet filter configuration without interrupting service.

Our framework simulates the traversal of packets through a distribution−point load−balancing device. As input we have used traffic generated from firewall logs for the University of Canterbury, spanning midnight, 25 July 2000 to the following midnight, as described in Section 12.2.1.

The scheduling models accept an input stream of packet details (including addressing, direction and connection number − essentially TCP/UDP and IP headers), and assign a node number to each packet. This is then fed into collating and graphing modules, the output of which is presented in Section 12.6. The individual scheduling models (refer to Section 9.3.2) used were:

- **Simple Round Robin** (SRR): Sequentially numbered connections are associated as *node = Connection number mod number of nodes*.
- **Dynamic NAT−friendly hash mapping** (BALANCE): *node = (source IP + destination IP + remote port) mod (number of nodes).*[65]
- **Least connection** (LC): Each new connection is allocated to the first node with fewest current connections, with connections removed after the final packet.
- **GLOB**: the model maintains a cluster−wide series of rules (as directed by the *lbmodel* processes, see 12.2), each associated with a node. Our model routes a packet to the node of a rule that matches that packet, and maintains rule and system counters.

We expect the behaviour of real systems to closely resemble that observed in this model environment. Issues such as packet loss, the effect of asymmetric routing techniques (for distribution mechanisms such as IP tunnelling), and cluster overload have not been explored, however.

12.2.1. Input traffic patterns

In order to evaluate the different scheduling models and the GLOB design, the model makes use of simulated traffic streams. These streams were generated using the external firewall logs of the University of Canterbury on 25 July 2000, spanning a 24−hour period.

An alternative technique, which was trialled but found unsatisfactory, was the use of randomly generated input. Accurately simulating the complex usage and protocol patterns found in real networks is a difficult problem, and would prejudice the results of any scheduling scheme evaluation. Basing the model input data based on observed

65 This is the same load−balancing hash used in the Netfilter BALANCE target.

network behaviour allows some confidence that the results obtained is representative of the behaviour of some real−world networks.

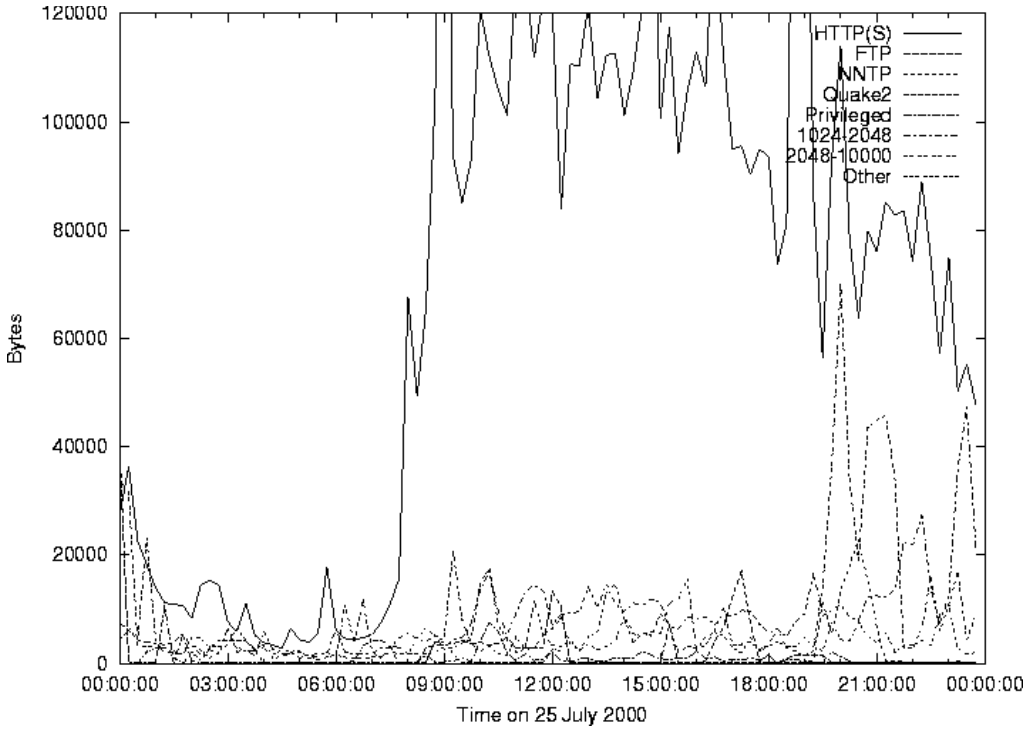

Figure 24. Protocol distribution of traffic based on remote port numbers

Figure 24 illustrates the variation in traffic consistency. In the trace, outgoing HTTP connections make up 73% of the total traffic load, while 55% of all outgoing connection traffic is directed at the University's external proxy server. Clearly, the observed traffic is not homogeneous across the possible IP address or port space. For a load balancing system that uses addressing details to share load, such as hash−based systems (eg. BALANCE scheduling model) or GLOB, this greatly complicates fair load distribution.

The initial log entries describe the source and destination addresses, packet and byte counts, and connections duration. These were mapped onto simple packet streams consisting a number of packets with alternating directions, equal packet intervals and payload sizes (with any excess added to the first packet). This does not fully model bursty or asymmetric connections, and does not represent protocol flow control and reliability features. Address and protocol distributions found in the base log are however represented realistically.

12.3. Lbmodel: the load−balancing core

On every node in the gateway cluster, an *lbmodel* component is responsible for

maintaining the load−balancing structure. It communicates with the central GLOB modelling component (when running a model) or with the Linux Netfilter structure to create and monitor rules, and communicates with peer *lbmodel* components to allocate and monitor the load distribution. To place the following discussion into context, a simplified example is presented in Section 12.3.4.

The simple initial state of the system is a single rule (segment of traffic defined by a packet filtering mask) that allows packets with any source and destination port through. By splitting and passing rules (see Figure 25), responsibility for traffic is shared among nodes, until all nodes have similar loads. Should one node fail, other nodes take over the load.

To achieve this, the *lbmodel* life cycle operates in the following manner:

1. Every *pollfreq* seconds, evaluate the current system state:

   (a) Clear out outdated rule entries held locally for remote rules. The *lbmodel* component tracks remote rules to improve local reporting and support evaluation of backup loads. If a rule has not been updated in a number of polling cycles, it is dropped to avoid inclusion of overlapping rules.

   (b) Extract load statistics for all rules (byte and packet counters) and the total traffic seen by that node from the GLOB model or Netfilter system. Rule loads are calculated as *new value = (old total * history factor) + (query load * (1 − history factor))*[66], for either packets or bytes.

   (c) Calculate node loads − total load for this node and backup loads for all other nodes. A node's backup load is the sum of all local rules that have that given node tagged as the backup. This represents the expected load on other nodes if the local device should fail − balancing this increases the chance of clean fail−over.

   (d) Split the highest load rule if required: if rule transfer in the previous poll failed, if there is only one local rule, or possibly based on some heuristic[67]. The decision to split a rule is often made in the previous polling cycle, as a result of being unable to shift a rule in the existing set.

   (e) Run the load−balancing component:

      i. For any rule that does not have a backup node, select the node with the lowest total (real + existing backup) load as backup (starting from the highest

66 This may be recognised as being the exponential average used in [TCP].

67 Initially, we split whenever the largest rule was at least *n* times larger than the next local rule. This seemed to lead to many unnecessary splits, however, and has been abandoned.

load rule down). If the current node fails, the backup node takes over this rule. Balancing total loads reduces the need for rebalancing immediately after a device crash. When nominating a node as a backup, a reliable message to this effect is sent. In addition, resumption state distribution may be necessary to support application−level fail−over.

ii. Decide whether this node is overloaded. The node is considered overloaded if its load was more, in proportion, than the average load by some threshold. For example, a node in a cluster of four devices that is handling more than 27.5% (10% more than ¼) of the total workload would be overloaded.

iii. If this node is overworked, attempt to find the rule that:

A. Has not been modified earlier in this poll. This ensures that changes to rules (and related state transfers to support application−level fail−over) have a chance to reach backup nodes before a rule is shifted.

B. Will not cause the destination node (= the backup for that rule) to become overloaded. Overloading another node causes cascading shifts, as that node shifts a rule to another node (or returns the overloading rule).

C. Will cause this node's load to approach the average load most closely. By returning to the ideal state, this node's contribution to the total system imbalance is minimized.

iv. If a suitable rule is found, transfer that to its backup node. This involves reliably notifying the backup node, and awaiting successful receipt. If successful, the backup and primary nodes for that rule are swapped. Otherwise, set a flag to require a split in the next poll, and force a rule backup nomination change.

v. Consider whether to change the backup node of a rule. Select the heaviest node (identified by backup + real load, backing up a local rule), and the lightest node overall. Locate the local rule that, if shifted, would decrease the difference between the average load and the total loads for the two nodes selected before by the greatest amount (optimising the system load in the event of this node's failure), and has not been recently modified. If modifying the backup improves the total system balance by more than a *backup margin* threshold, or if the rule shift failed, change the backup node for that rule to the lightest node.

(f) Broadcast state messages to all other *lbmodel* instances, including the system

load, and a list of all local rule states. The system load included in this broadcast is the local load relative to the total observed load − this compensates for time differences between when the load figure is calculated and used, which can be more than a full polling cycle period.

2. Between polls, the system sleeps and accepts messages from peers, updating its internal state. Depending on the frequency of polling episodes, the lbmodel should require minimal processing resources on the firewall host nodes.

When modifying the rule set, messages are also sent to any other nodes involved − generally up to two backup nodes, old and new − to update their internal states. These messages generally require reliable delivery and, in the case of a rule transfer, must be completed in a short (finite) amount of time. In addition, delivery of the messages must succeed or fail, but may not be ambiguous.

12.3.1. Rule splitting

In order to ensure that the loads between nodes are balanced, it is necessary to have a complementary set of rules that can be split equally (by load) between all nodes. Since the optimal split for a given request stream is unknown beforehand (and may not be static), a rule set covering all possible divisions would have to consist of many rules.

An alternative technique is to monitor the load represented by rules, and to divide heavily loaded rules into less loaded subrules. Repeated splitting allows heterogeneous traffic distribution across the rule space − the worst case for static rule sets, where large load concentrations fall into small address ranges − to be reduced to rules with roughly equal average loads, via a binary decomposition.

In order to split a rule, the intervals represented by the internal and external port ranges are calculated, and the larger interval divided down the middle. Since the system cannot predict which part of a split will be more heavily loaded, unequal splits do not speed up the process.

In addition, the traffic load history associated with that rule is divided between the halves, the rules are tagged as modified and the GLOB model or Netfilter system is modified. Finally, the backup node shared between both halves is reliably notified of the change.

12.3.2. Rule transfer

The basis of load−balancing in the GLOB system lies in its ability to transfer excess work from node to node. Transferring a rule from one node to its backup consists of the following steps:

- Disable the rule in the underlying local packet filtering system. Any packets associated with that rule will now be dropped. To avoid dropped connections, the transfer process must complete or fail within the timeout period of protocols carried across this link.
- Notify the backup node that it should take over the rule. In the model system, Unix domain sockets are used, guaranteeing reliable delivery. On a network implementation, some reliable delivery mechanism would be required, capable of completing or failing within a set, short period. We know of no technique for absolutely guaranteeing this requirement, but describe a technique offering statistical guarantees in Appendix 1.
- If the rule transfer message is received successfully, note that the source (overloaded) node is now the backup for that rule, and vice versa. The remote backup node then enables the rule on the underlying packet filter, and starts handling packets.
- If the delivery fails, reinstate the rule locally.

Note that, as part of the rule transfer process, any connections currently associated with that rule are also transferred. This requires the receiving node to have a mechanism in place for handling such existing connections − either by passively tracking state for all rules it backs up (in the form of a two−stage filter, allowing both local and backup traffic in, but only responses to local traffic out), or by having an explicit state−transfer mechanism supporting the rule transfer[68]. Refer to Section 7 for a more detailed discussion on tracking options. This also implies that changing the backup node for a rule is an expensive operation. The new backup node must acquire sufficient state to support rule transfers, passively or explicitly, within a polling cycle.

12.3.3. Node fail−over

Some form of heartbeat mechanism is required to identify node failure, as described in Section 9. This can be done internally, by tracking node state notifications, or by

68 Noted that the two−stage filter mechanism effectively creates a one−node backup to the rule. For a two−node system, this is simply a variation of a hot−standby configuration; for more nodes it scales up accordingly.

an external system such as *mon* [MON] or *Piranha* [Piranha]]. The latter approach is currently envisaged: in order to identify node failure quickly enough to allow connections to survive, the polling frequency would have to be consistently high, consuming network bandwidth and node processing time.

In the model, any *lbmodel* instance can be instructed to self−destruct. This causes it to immediately remove all of its rules from the GLOB model framework (simplifying cleanup), broadcast a message to all other *lbmodel* instances, and terminate. In a production system the cleanup stage would be unnecessary, and the heartbeat mechanism need only broadcast a message to the remaining node monitors.

An *lbmodel* component, on receiving a termination message for another node, proceeds to mark all rules of that node for which it is the backup as local (effectively transferring all rules from the failed node to their backups), and updates the underlying packet filter. If an *lbmodel* component should receive a termination message for itself, it must yield all its rules and terminate. This allows nodes to be explicitly removed from the cluster, and terminally overloaded or unstable nodes (to the point where this node no longer responds to the heartbeat) to recover.

12.3.4. A simplified example

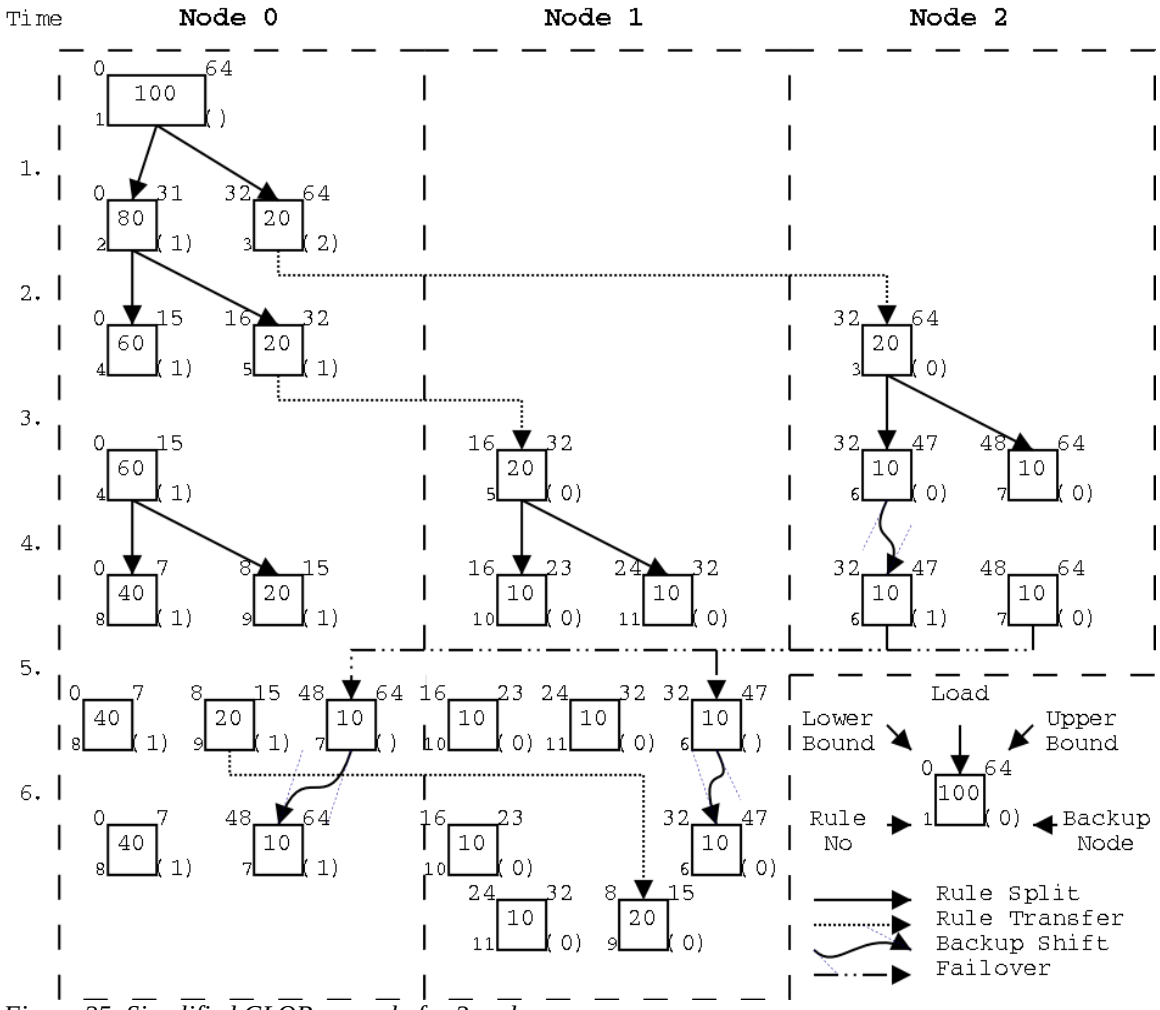

Figure 25. Simplified GLOB example for 3 nodes

To illustrate the functioning of the GLOB system, consider a simplified example. This example uses a fixed single port range of 0−64, and simple fixed rule loads. As shown in Figure 25, the following events occur:

0. At the start of the model, node 0 has a single rule covering the entire port space, with an arbitrary load of 100. Nodes 1 and 2 have no associated rules or loads.
1. At the end of the first polling cycle, node 0 executes the process outlined in Section 12.2, with the following results:
   - Node 0 has a single rule (rule 1), so this is split into rules 2 and 3. The load observation during the next polling interval will serve to correct the rule loads − to simplify matters we assume this correction will take a single interval.
   - Rules 2 and 3 have no backup nodes assigned, so node 1 is assigned as backup to rule 2 and node 2 as backup for rule 3.

Though node 1 is overloaded, it cannot transfer any load to other nodes, since all local rules have been modified. Instead, it will execute another rule split early in the next poll.

2. At the second poll, node 0 is still has serves the entire load. It executes the following actions:
   - Rule 2 is split into rules 4 and 5, since node 0 failed to transfer any load in the previous poll.
   - Node 0 is still overloaded, so it shifts rule 3 to node 2 (the backup for that rule). Note that rules 4 and 5 could not be shifted, since they had been modified in this cycle.
3. In the third poll, both nodes 0 and 2 have loads, resulting in actions:
   - Node 0 transfers rule 5 to node 1. Since it transferred a rule in the previous poll, and had two rules, it does not split.
   - Node 2 splits its single rule into rules 6 and 7. Note that after the transfer, node 0 is the backup for rule 3, and hence for rules 6 and 7.
4. The fourth poll has all three nodes loaded, with node 0 still overloaded. The following actions take place:
   - Node 0 splits its single rule 4 into rules 8 and 9.
   - Node 1 splits its single rule 5 into rules 10 and 11.
   - Node 2 is not overloaded and has multiple rules, so no split or transfer takes place. The heaviest node used as backup here is node 0 (total load 80), with the lightest node (excluding 2 itself) being node 1 with total load 20. Sharing the load 70:30 is significantly better than a 80:20 split, so rule 6's backup node is changed to 1.
5. At this time, node 2 fails. The remaining nodes each take over the rules that they are backup nodes for (node 0 taking rule 7 and node 1 taking rule 6). After this process, these rules no longer have backup nodes allocated.
6. At the fifth poll, only two nodes remain. Of these, node 0 is still overloaded, and rules 6 and 7 have no backup nodes. Node 0 selects node 1 as backup for rule 7, and transfers rule 9 to that node. Node 1 selects node 0 as backup for rule 6.

   Note that node 0 could not select to transfer rule 8, since that would overload node 1. Neither could it transfer rule 7, since it had been modified earlier in the poll. Even if this was not the case, rule 9's load most closely matches the difference

between node 0's load and the perfect load of 50.

The system would then stabilise, since rule loads are stable in this example. In a real system, the rule loads change from poll to poll (using a geometric sliding scale), requiring constant correction to maintain an ideal balance. Similarly, in a real system, the 0−64 port interval would be replaced by the full TCP/UDP source and destination port (and possibly IP) address space.

12.4. Interaction with Firewalls

The GLOB system was designed to allow general network software to be layered on top of it. In particular, firewall packet filtering and proxy facilities are supported, along with some forms of NAT[69]. In particular, packets reaching a GLOB node (running a firewall) pass through the following stages:

1. A packet is received and sanity−checked: valid checksums, correct IP headers, etc.
2. Incoming packets are passed through NAT to produce usable internal addresses.
3. The packet is passed through the local GLOB rules. Packets that are the responsibility of other nodes are discarded.
4. The packet is passed through the conventional, possibly stateful, packet filtering system present on the firewall or server. The filtering rules for this are exactly as they would be for a single−node system, including connection tracking.
5. The packet is passed to local processes (service software or proxies) for handling if necessary.
6. Local responses and routed packets are passed back through the firewall packet filters.
7. Outgoing packets are passed through the local GLOB rules. If two−stage filtering is used (to support slave application processes, see Section 7.5.2), packets matching backup rules are dropped.
8. Outgoing packets are passed through NAT to map private addresses to external representations, if necessary.
9. The packet is transmitted onto the appropriate network link.

As can be seen, GLOB functions as an initial packet filter. Otherwise, it should have no impact on node functions, being effectively transparent. The packet filter used in

69 As with conventional packet filters, GLOB is complicated by NAT address changes. One way to support NAT is to require that packets presented to the GLOB filters must have correct internal and external addresses. Since this requires NAT, and connection tracking, to take place before load sharing, this reduces the value of the load−balancing system significantly.

GLOB is completely independent, and precedes, any packet filter used on the firewall[70].

12.5. Extensions

The current GLOB system is useful as a proof−of−concept model. A number of modifications are under consideration, which would improve the load−balancing ability, handling of complex protocols, and network implementation of the system.

12.5.1. A fast, unequivocal delivery algorithm

In order to maintain the unique mapping of packet to node, the GLOB system must ensure that all possible packets are associated with some rule (i.e. no rules are lost), and that no overlap between rules exists. Failure to do so leads to connection failures or system duplication and instability.

When a rule transfer takes place, either the originating node or the destination must then take final possession of the rule. This process must be completed, whether successful or not, within a finite short period. In particular, the process must finish within the time that affected connections accept as transitive failures. As described in Appendix 1, this is not trivial − we present a possible approximate solution there.

12.5.2. Dynamic internal configuration

Currently, GLOB only splits on port ranges, and has fixed polling intervals. Adding splitting by IP should markedly improve the ability of the system to balance loads quickly, while improving support for complex protocols. By extending or shrinking the polling interval, based on a node's recent loading, the overhead imposed by polling − and, in particular, by rule state changes − could be minimized.

In addition, differentiating between TCP and UDP traffic could optimise the different traffic characteristics of these protocols. Similarly, support for ICMP and multicast protocols could also be included, via address splitting and predefined rules.[71]

Separating packet streams for which no higher−level state is kept from state−bound connections would allow rule transfers to avoid state−bound rules whenever possible, reducing the level of pre−emption used. However, since effective firewall security

70 Applying the GLOB rules after firewall packet filtering rules may be useful when the additional load represented by firewall rules is small − as may be the case when NAT or a simple restrictive static filtering rule set is in use.

71 ICMP error messages carry the IP header and 64 bytes of message body. These messages should be filtered based on these values, rather than the outer IP and ICMP headers (as with any packet filter).

increasingly requires stateful tracking of connectionless protocols, this optimisation may not be as effective in practice.

12.5.3. Rule remerging

When splitting rules to balance heterogeneous loads, each split typically produces one subrule with a large load, and one with minimal load. The heavily loaded rule is then split again, increasing the number of unloaded rules. By merging such rules, the overall number of rules in the system can be reduced (reducing packet filtering and state update overheads), while minimally affecting the ability of the system to rebalance.

12.5.4. Service−specific and bound rules

Many modern network services rely on multiple connections, such as FTP and session−based HTTP. In addition, connections to different services from the same client (such as HTTP and HTTPS) may require access to the same service node, to allow session state maintenance.

GLOB supports such protocols by defining static rules identifying all possible port combinations, for example [20,21 −> 1024−65535 for a passive FTP server][72], and tagging such rules to prevent port splits. The current rule structure does not support linking of rules or port sets (as opposed to port intervals), an extension that could be added. Such complex services would be balanced by using address splits, avoiding the possibility of associated connections running across multiple rules.

12.5.5. State transfer systems

As noted in Section 12.3.2, rule transfers depend on the new owner having sufficient state information for current requests. We have designed a proxy framework which explores aspects of this for proxying firewalls; for general−purpose servers, the two−stage filtering mechanism described earlier should be explored further.

Simply put, a two−stage filter accepts any incoming packets matching either a local or a backup rule, but only transmits matching local rules. For connection tracking systems (such as stateful packet filters or NAT systems), this allows the backup node to maintain its internal state. Any system that relies on local decision making during state tracking − such as NAT port allocation, or proxy connection sequence number selection − would have to be modified to produce the same results between primary

72 These static rules are the same as the rules used in a static packet filter to support such complex protocols.

and backup.

We currently have prototype state transfer−capable proxies capable of resuming connections from the last checkpoint. Work is in progress to extend this into fully resuming connections, without loss of data in transit, as covered in Section 7.

12.5.6. Non−disruptive Rule Transfer

Generally, when a GLOB rule is transferred, it changes the path of any connections that fall under that rule. Ideally, the new rule host can resume processing of any such connections seamlessly.

When using general network applications without fail−over support, however, this behaviour would lead to frequent service disruption. In addition, even when such capabilities are present in higher−level systems, resuming connections can be computationally expensive. In these cases, it would be preferable to shift rules that do not currently carry connections.

Extending the lbmodel rule monitoring capabilities to include whether a given rule is in use would require connection−tracking capabilities. This tracking would only be necessary for connections covered by the local rule set, maintaining load−balancing and scalability. Ideally, the connection tracking system already present in the Netfilter structure could be used.

In the model framework, a comparison was made between non−pre−emptive scheduling methods and a pre−emptive GLOB system. While such a comparison shows the potential of the GLOB technique, a more fair comparison would include the effects of pre−emption.

12.6. Results

Of the data stream described in Section 12.2.1, data for the period 11:00−11:15 am was used to evaluate the effect of different scheduling schemes on load−balancing quality.[73] Figure 24 illustrate the variation in protocol and addressing spread present in this extract.

This packet stream was directed through each of the models (see Section 12.2) in turn, with varying cluster sizes, and post−processed to generate the statistics and graphs shown here. The GLOB system was run with an inter−poll interval of 6 seconds real time, a load threshold of 1.1 (so if the node load exceeds the average by

73 This period was arbitrarily chosen − any time segment in the 24−hour period could be used. During this period, a total of 126MB of traffic was handled.

10% it is overloaded), and a history factor of 1/3 (controlling the exponential averaging process).

The columns shown in Table 3 represent:

- **# Nodes** − the number of nodes in the service cluster.
- **Model** − which scheduling method was used to generate the listed values.
- **Full period** − these sets of statistics apply to the full measurement period.
  - **Peak load (Bps)** − the highest bandwidth allocated to any node during a 10s interval.
  - **Load StdDev** − the standard deviation of node loads relative to the average load, from the exact average load of 1. About 68% of node load observations are expected to fall within *±1 StdDev* of this value, 95% within twice that. [Stat]
  - **Peak load (fraction)** − the highest share of the offered load handled by any one node, relative to the expected proportion.
- **Stable period** − these statistics were calculated without the initial 180s period of the model run. This allows the GLOB learning phase to complete, offering a more realistic long−term performance comparison.

# Nodes	*Model*	*Full Period*			*Stable Period*		
		Peak load (Bps)	*Load StdDev*	*Peak load (frac)*	*Peak load (Bps)*	*Load StdDev*	*Peak load (frac)*
3	Simple RR	89583	0.1591	1.456	83386	0.1500	1.456
3	BALANCE	89104	0.2289	1.696	79544	0.2112	1.696
3	Least Conn	89503	0.1813	1.521	89207	0.1725	1.521
3	GLOB	130032	0.2540	2.946	73624	0.1025	1.391
4	Simple RR	72626	0.2417	1.848	72626	0.2508	1.848
4	BALANCE	80013	0.3043	1.721	80013	0.3155	1.721
4	Least Conn	81968	0.2124	1.756	65661	0.1791	1.605
4	GLOB	130093	0.3227	3.930	52163	0.0888	1.278
6	Simple RR	59436	0.2353	1.983	48140	0.2221	1.864
6	BALANCE	62122	0.3889	2.310	57366	0.3881	2.310
6	Least Conn	60208	0.2621	2.009	52819	0.2457	1.910
6	GLOB	130635	0.5199	5.919	45734	0.1086	1.529
8	Simple RR	46322	0.3229	2.770	46322	0.3245	2.770
8	BALANCE	58520	0.5111	2.684	50834	0.4972	2.684

# Nodes	Model	Full Period			Stable Period		
		Peak load (Bps)	Load StdDev	Peak load (frac)	Peak load (Bps)	Load StdDev	Peak load (frac)
8	Least Conn	50169	0.3285	2.382	42591	0.3094	2.382
8	GLOB	130660	0.6470	7.893	33492	0.1287	1.590
12	Simple RR	34136	0.3752	2.978	34136	0.3724	2.978
12	BALANCE	45580	0.5644	3.112	45580	0.5646	3.112
12	Least Conn	41384	0.3956	2.999	35996	0.3661	2.683
12	GLOB	131873	0.8792	11.844	25796	0.2291	2.055
16	Simple RR	34085	0.4629	4.077	34085	0.4548	4.077
16	BALANCE	40062	0.6289	3.690	31639	0.6065	3.383
16	Least Conn	36256	0.4589	4.035	36256	0.4408	4.035
16	GLOB	130691	1.0822	15.790	26215	0.3034	2.741

Table 12. Scheduling method evaluation statistics

From Table 12, we can draw a number of conclusions:

- On scheduling models other than GLOB, there is little difference between observations for the full and stable periods. This is expected, since these methods have no learning capabilities.
- When considering the full period, the peak values of the GLOB model are relatively constant. From load graphs, however, it is clear that these values are artefacts of the initial learning phase for GLOB. It is therefore reasonable to consider not only statistics derived from the full period, but also those generated after the learning phase.

  Currently, the GLOB system uses an initial state where the first node accepts the full incoming load. This resolves problems with synchronising the initialisation of a cluster, but may lead to overload problems. To avoid this, it is possible to use a different initial state − candidates include some heuristic rule set (based on common services), a previously learned rule set, or a simple load division[74].
- The results for a number of methods are surprisingly counter−intuitive. In particular, the performance of Least Connection scheduling closely matches that of Simple Round Robin.
- As the number of nodes increase, the relative quality of load−balancing decreases.

74 GLOB can easily simulate a ONE−IP hash by using an appropriate rule set and round−robin rule allocation.

Splitting a fixed input stream into increasingly small segments is more sensitive to traffic anomalies. One implication of this is that, for large clusters serving highly variable request sizes, better quality balancing schedules are vital.

For example, in the 16−node cluster test (Table 12), the standard deviation figures produced are around 0.5. This means that we could (approximately) expect the node loads to be within twice that deviation of the expected average 95% of the time − implying that individual nodes require noticeably more processing power than expected. This, for a large number of nodes, implies a significant unnecessary investment.

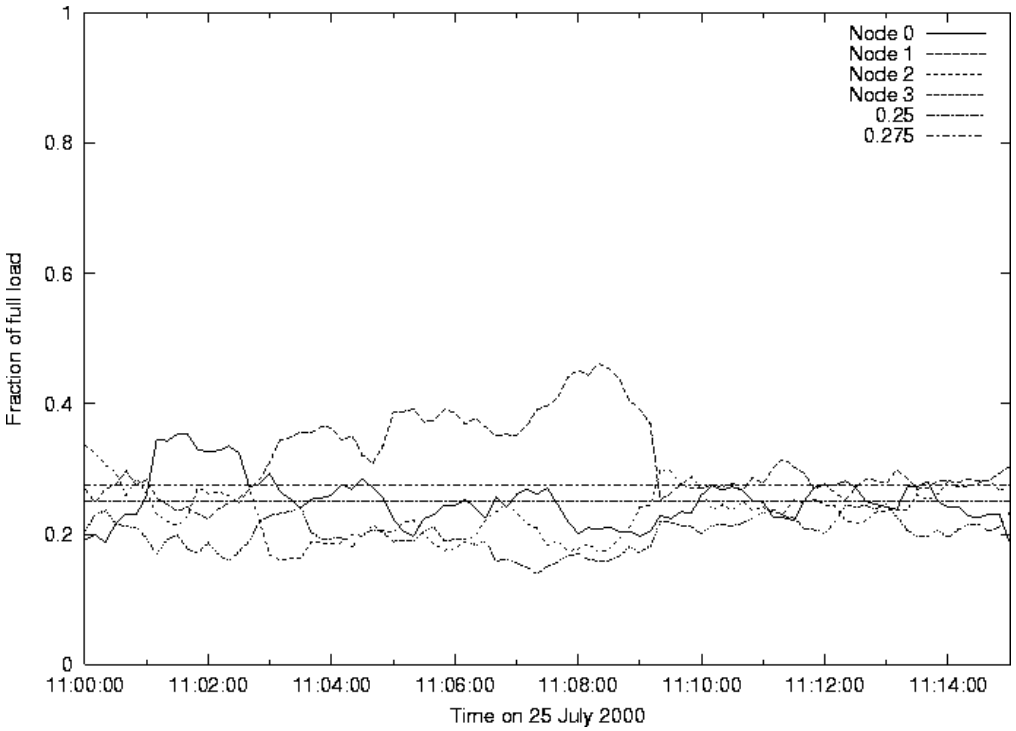

Figure 26. Simple Round Robin 4−node model relative load for 11:00−11:15

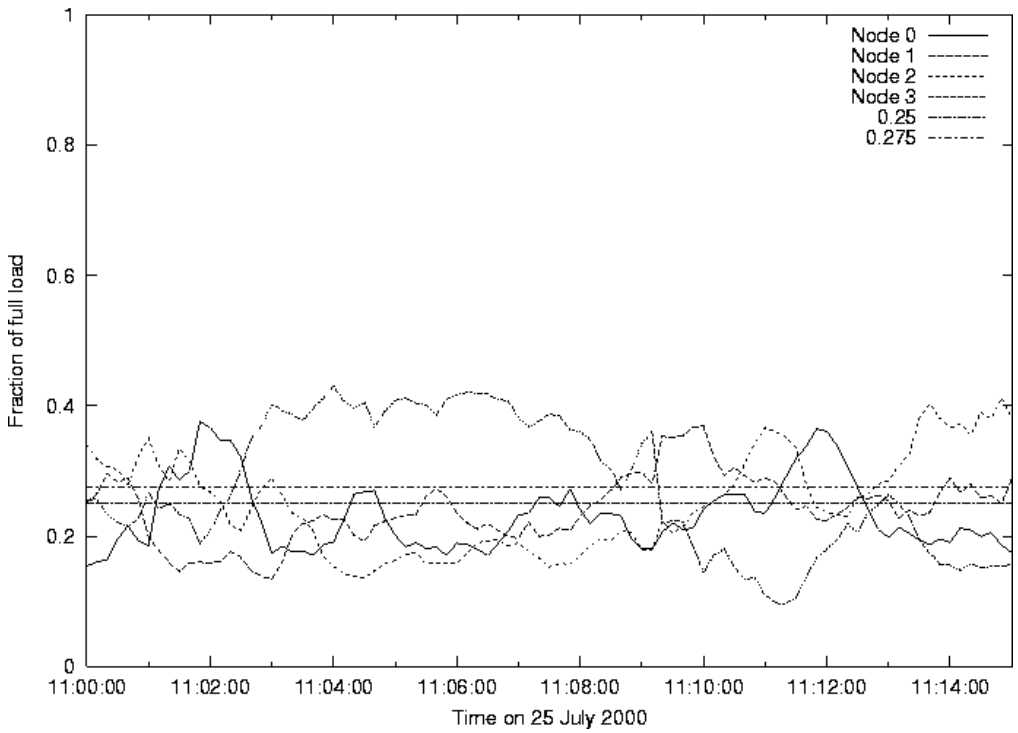

Figure 27. Netfilter BALANCE 4−node model relative load for 11:00−11:15

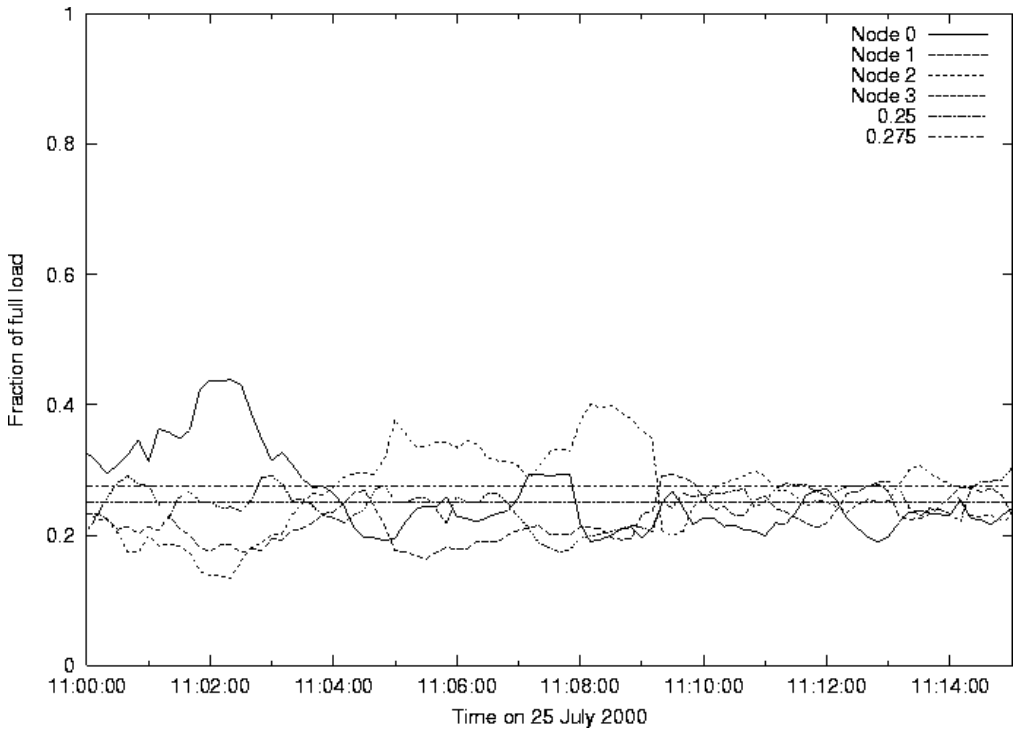

Figure 28. Least Connections 4−node model relative load for 11:00−11:15

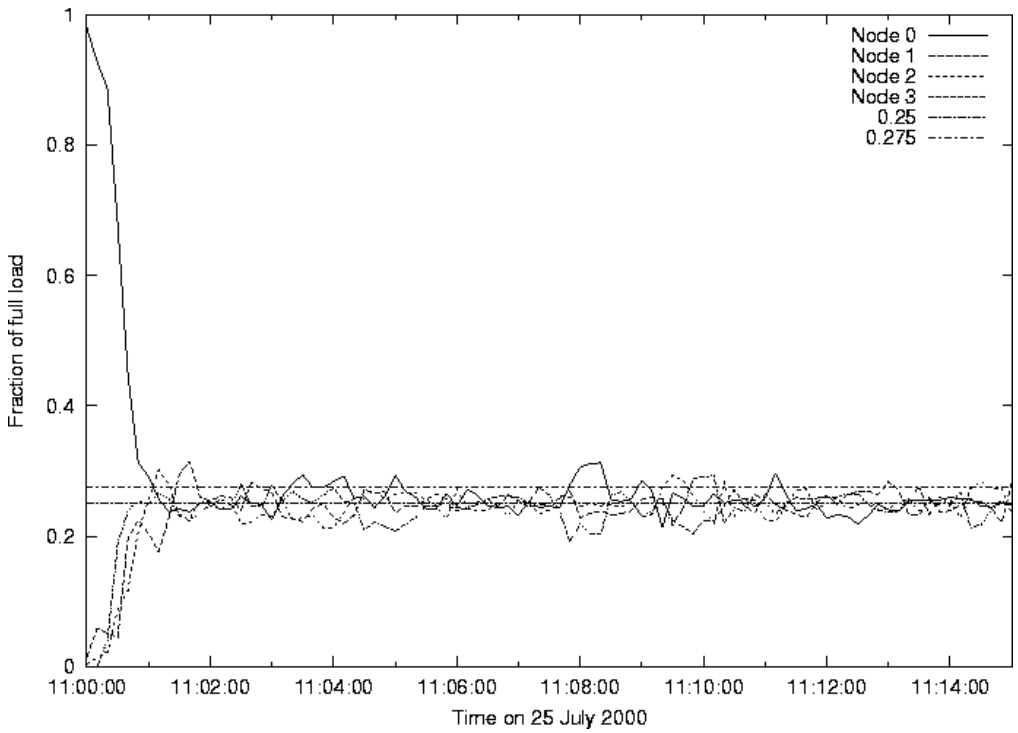

Figure 29. GLOB 4−node model relative load for 11:00−11:15

As can be seen from Figure 24, the input stream is not smooth. Therefore, most of the graphs are presented in terms of relative load − the fraction of the total load each node is handling. In addition, there are horizontal lines on such graphs showing the expected (average) load (*=1/number of nodes*), and the GLOB correction threshold.

The model output (Figures 26, 27, 28 and 29 and Table 12) show a clear hierarchy in the effectiveness of the models used: the Netfilter BALANCE hash performs poorly, followed by Simple Round Robin. Least Connections appears to perform slightly better, with GLOB following the optimal loads closely, after an initial learning phase.

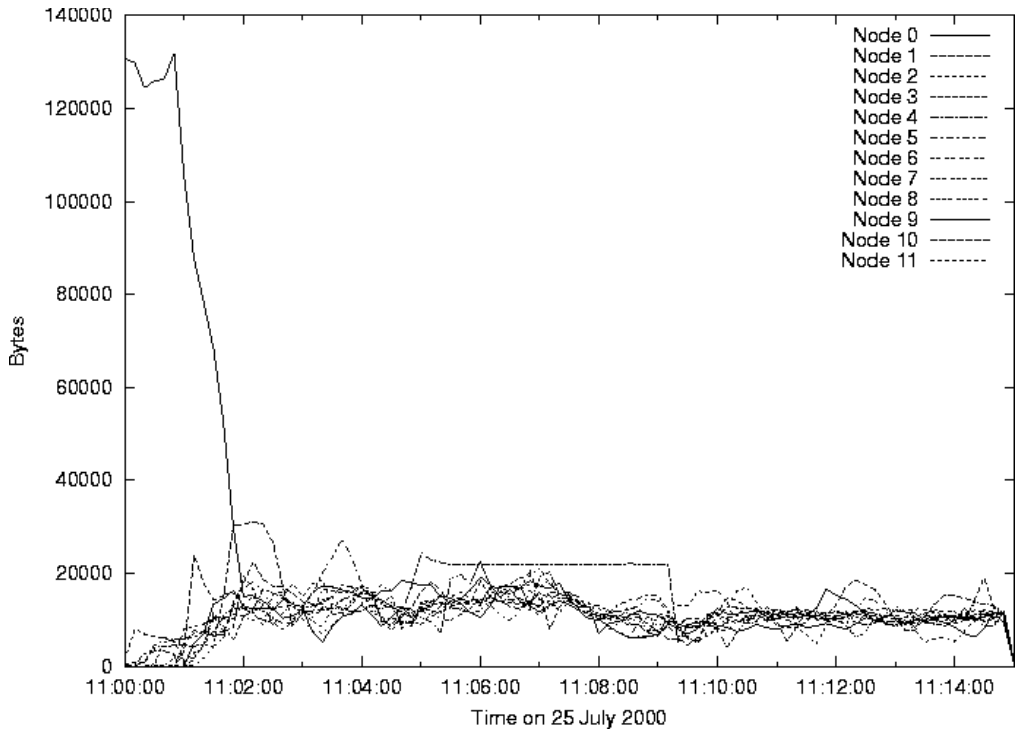

Figure 30. GLOB 12−node model load for 11:00−11:15

The graph in Figure 30 shows the result of applying GLOB to a larger number of nodes. Note the extended learning phase, and the number of spikes that self−correct. An interesting anomaly in this graph is the section from about about 11:04−11:10, where a traffic increase does not appear to be corrected. On inspecting the input trace, however, we found that this represents a single HTTP connection, transferring about 5mb over the course of 245 seconds − a data rate of about 23 kb/s. What happens is that the GLOB system on that node sheds as much traffic as possible, until it is reduced to handling only that connection. When that connection terminates, it is relatively lightly loaded. In the following polling cycles, this node then reabsorbs duties from the rest of the cluster.

12.7. Conclusion

In this section, a new method of load−balancing using adaptive filtering techniques has been discussed. Preliminary results show that this method has the potential to significantly improve the fairness of load−balancing on a cluster over current methods. In addition, the GLOB system presented is capable of use in situations where current alternatives fall short, especially in its capability to handle multihomed systems. An important application of the GLOB architecture is in the development of a load−balanced firewall system.

A great deal of work remains, however − in extending the capabilities of the GLOB method, in evaluating these and other models over longer periods, and in building the state−transfer mechanisms underlying GLOB. In addition, development of a GLOB variant that does not require state transfer would improve the general application of this method significantly. Finally, there are a number of optimisations possible, and a variety ipcof internal settings in the GLOB algorithm to be explored.

13. Conclusion

This thesis has focused on alleviating the limitations of current security techniques. Conventional monolithic firewalls form a single point of failure in networks, do not scale well, and are often prohibitively expensive. In addition, modern firewalling is a trade−off between fast packet filtering techniques without fine−grained control, and more powerful application proxies with high processing overheads.

Conventional firewalls have also proved insufficient to protect public network services against attack. Of the other security techniques available, none resolve this problem. Host and application security constantly lags behind newly discovered flaws, and require constant maintenance of systems. Cryptography does not protect against untrusted peers. Intrusion detection systems, while offering awareness of attacks, is a responsive rather than preventative measure.

Hybrid intrusion detection and firewall systems have the potential to protect against application−level attacks. These systems have even greater performance limitations than proxy firewalls, however.

The goal of this thesis was to create a prototype firewall architecture that addresses the processing, point of failure and scalability problems of monolithic conventional or hybrid firewalls. By sharing the processing load across a number of firewall nodes in a cluster configuration, a firewall has access to greatly extensible resources.

With this in mind, this thesis first presented a design for such a system, consisting of a load−balancing foundation, heartbeat mechanisms to link nodes into a fail−over capable system, and an application proxy framework supporting failure recovery.

The proxy framework, with the supporting *prism* connection recovery module, is a simple extensible structure for transparent protocol−aware application proxies. Prototype implementations of this framework have demonstrated its applicability to real protocols, and shown some capability for restarting connections. In particular, connections that do not have data transfer during the failure recovery period can be fully recovered, while other connections are subject to data loss. Section 7.5 also presents techniques possible to resolve this data loss, resulting in completely recoverable proxies.

Heartbeat mechanisms have been widely implemented in existing failover systems. These systems are typically oriented towards singular master−slave links, however. Section 4.2.3 outlines mechanisms for extending this to a peer network, in particular

taking advantage of the broadcast medium shared between nodes.

The final section presents a prototype of the GLOB (Generic Load Balancing) system. Unlike alternative schemes (described in Section 9), this mechanism has been specifically designed to support clusters connected to multiple networks. Section 12.6 presents results obtained from a model implementation of GLOB, in comparison to popular scheduling schemes. These results indicate that the GLOB scheme has the potential to significantly improve the load balancing quality over competing mechanisms, without introducing single points of failure or breaking transparency.

We hope to develop a future version of GLOB that does not rely on connection preemption, and to implement the system on a real network. In addition, a number of extensions to the GLOB mechanism are yet to be explored − support for complex services, full fail−over implementation (including application−level structures), and improvements to the general mechanism.

Finally, we want to to combine all of the components developed to implement a fully distributed, fail−over capable, scalable firewall system supporting hybrid IDS and proxy applications, in addition to conventional firewalling techniques.

14. References

[0−Frag] John McDonald "DoS for Linux 2.1.89 − 2.2.3: 0 length fragment bug", *Bugtraq mailing list*, March 1999, http://www.securityfocus.com/frames/?content=/templates/archive.pike%3Flist%3D1%26mid%3D12958%26_ref%3D1900797774

[AAFID] CERIAS Autonomous Agents for Intrusion Detection Group, "COAST Autonomous Agents for Intrusion Detection", http://www.cerias.purdue.edu/homes/aafid/

[Aleph1] Aleph One, "Smashing The Stack For Fun And Profit", *Phrack Magazine, Vol.7 Issue 49 File 14/16*, http://www.codetalker.com/whitepapers/other/p49−14.html

[ANS] Theuns Verwoerd, "Active Network Security", *University of Canterbury BSc Hons Project*, November 1999, http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/1999/hons_9909.pdf

[ARP] D.C. Plummer "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware", *RFC 826*, November 1982, http://www.ietf.org/rfc/rfc826.txt

[ARPAnet] Eric C. Rosen, "Vulnerabilities of Network Control Protocols: An Example", *RFC 789*, http://www.ietf.org/rfc/rfc0789.txt

[AssgnNo] J. Reynolds, J. Postel "Assigned Numbers", *RFC 1700*, October 1994, http://www.ietf.org/rfc/rfc1700.txt

[BackDoor] CERT, "Interbase Server Contains Compiled−in Back Door Account", *CERT Advisory CA−2001−01*, January 2001, http://www.cert.org/advisories/CA−2001−01.html

[BackOrifice] Cult of the Dead Cow, "Worst Case Scenario", July 1999, http://www.cultdeadcow.com/tools/

[Bartley] Richard Bartley, "Corporate Information Security Strategy − how to avoid giving free information to attackers", *Xineta Ltd.*, March 2001, http://www.xinetica.com/tech_explained/general/avoid_giving_free_info/wp_avoid_giving_free_info.htm

[Bellovin] Steven M. Bellovin, William R. Cheswick, "Firewalls and Internet Security − Repelling the Wily Hacker", *Addison−Wesley ISBN 0−201−63357−4*, 1994

[Broadband] "Broadband Today", *Broadband Today 2000*, http://www.broadband−today.com/may2000.html

[Castle] E. Genise, "Rock of Cashel", *Web site*, http://mysite.ciaoweb.it/edgenis/cashel1.htm

[CCHack] Will Knight, "Hackers steal one million credit cards", *ZDNet UK*, March 2001, http://www.zdnet.co.uk/news/2001/9/ns−21473.html

[CHA] "High Availability Module − Secure Virtual Network Architecture", *Check Point Software Technologies*, 2001, http://www.checkpoint.com/products/vpn1/ha.html

[CIDR] V. Fuller, T. Li, J. Yu, K. Varadhan "Classless Inter−Domain Routing (CIDR): an Address Assignment and Aggregation Strategy", *RFC 1519*, September 1993, http://www.ietf.org/rfc/rfc1519.txt

[CSI] Richard Power, "2001 CSI/FBI Computer Crime and Security Survey", *Computer Security Institute, Computer Security Issues & Trends vol. VII no. 1*, Spring 2001, http://www.gocsi.com/prelea_000321.htm

[DDos] Dave Andersen, "Distributed Denial Of Service Attacks (DDOS)", February 2000, http://wind.lcs.mit.edu/~dga/ddos.txt

[DDoSFaq] Kurt Seifried, "Denial of Service (DoS) FAQ", *SecurityPortal.com*, March 2000, http://www.securityportal.com/research/ddosfaq.html

[Denning] Dorothy E. Denning, "Information Warfare and Security", *Addison−Wesley ISBN 0−201−43303−6*, 1999

[DFP] "The Cisco Dynamic Feedback Protocol", *Cisco Systems Inc.*, November 2000, http://www.cisco.com/warp/public/cc/pd/ibsw/mulb/tech/dfp_wp.htm

[DistFW] S. Ioannidis, A. Keromytis, S. Bellovin, J. Smith. "Implementing a Distributed Firewall" *Proceedings of Computer and Communications Security (CCS) 2000*, November 2000, http://www.cis.upenn.edu/~angelos/Papers/df.ps.gz

[DNS] P.V. Mockapetris "Domain Names − implementation and specification", *RFC 1035*, Noverber 1987, http://www.ietf.org/rfc/rfc1035.txt

[DOD] Sam Cox, Ron Stimaere, Tim Dean, Brad Ashley, "Information Assurance − the Achilles' Heel of Joint Vision 2010?", March 1999 http://www.airpower.maxwell.af.mil/airchronicles/cc/ashley.html

[Doorn] Leendert van Doorn, "Computer Break−ins: A Case Study", June 1994, ftp://coast.cs.purdue.edu/pub/doc/misc/Leendert_van_Doorn−Computer_Brakins.ps.Z

[DynDNS] David E. Smith, "Dynamic DNS", *Technopagan.org*, June 2001, http://www.technopagan.org/dynamic/

[eCommerce] "Despite Customer Service Woes, Online Retailing Keeps Growing", *internet.com CyberAtlas*, March 2001, http://cyberatlas.internet.com/markets/retailing/article/1,1323,6061_719771,00.html

[Emerald] Phillip A. Porras, Peter G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Distrubances", 1997 National Information Systems Security Conference, 1997, http://www.sdl.sri.com/emerald/Emeralkd−NISS97.ps.gz

[fake] "Fake: IP Address Takeover Tool", http://www.ca.us.vergenet.net/linux/fake/

[Forensic] Lance Spitzner, "Know Your Enemy: A Forensic Analysis", May 2000, http://www.enteract.com/~lspitz/forensics/

[FragSec] G. Ziemba, D. Reed, P. Traina "Security Considerations for IP Fragment Filtering", *RFC 1858*, October 1995, http://www.ietf.org/rfc/rfc1858.txt

[FTP] J. Postel, J.K. Reynolds "File Transfer Protocol", *RFC 959*, October 1985, http://www.ietf.org/rfc/rfc0959.txt

[FTPBounce] CERT, "FTP Bounce", *CERT Advisory CA−1997−27*, March 1999, http://www.cert.org/advisories/CA−1997−27.html

[FW−1] "Check Point Firewall−1", *Check Point Software Technologies*, http://www.checkpoint.com/products/firewall−1/index.html

[FW−1Eval] Thomas Lopatic, John McDonald, Dug Song, "A Stateful Inspection of Firewall−1", *Black Hat Briefings 2000*, August 2000, http://www.dataprotect.com/bh2000/blackhat−fw1.html

[FWLB] Vik Varma, "A Solution to Redundant, Load Balanced Firewall Systems", November 1998, http://www.netwizards.net/~varmav/tips−tools/firewall_paper.shtml

[FWList] "Firewall Product Selector", *CSI*, http://www.spirit.com/cgi−new/report.pl?dbase=fw&function=view

[FWReview] Keith Schultz, "Enterprise Firewalls − Firewalls Go Gigabit", *CPM TechWeb Internetweek*, March 2001, http://www.internetweek.com/reviews01/rev032601.htm

[FWRFC] N. Freed, "Behavior of and Requirements for Internet Firewalls", *RFC 2979*, October 2000, http://www.ietf.org/rfc/rfc2979.txt

[FWTK] "TIS Internet Firewall Toolkit", *Trusted Information Systems, Inc.*, http://www.tis.com/research/software/

[GPL] "GNU General Public License", *Free Software Foundation*, July 2000, http://www.gnu.org/copyleft/gpl.html

[GRCDoS] Steve Gibson, "The Strange Tale of the Denial of Service Attacks Against GRC.COM", *Gibson Research Corporation*, May 2001 http://grc.com/dos/grcdos.htm

[GrIDS] S. Cheung *et al*, "The Design of GrIDS: A Graph−Based Intrusion Detection System", *U.C. Davis Computer Science Department Technical Report CSE−99−2*, 1999, http://seclab.cs.ucdavis.edu/arpa/grids/grids.pdf

[HackWar] Jonah Greenberg, "Chinese hackers declare 'May Day War'", *ZDNet News*, May 2001, http://www.zdnet.com/zdnn/stories/news/0,4586,2714179,00.html

[Hobbit] Hobbit, "CIFS: Common Insecurities Fail Scrutiny", January 1997, ftp://ftp.technotronic.com/rfc/cifs.txt

[Honeynet] "The Honeynet Project", http://project.honeynet.org/

[HSRP] T. Li, B. Cole, P. Morton, D. Li, "Cisco Hot Standby Router Protocol (HSRP)",*RFC 2281*, March 1998, http://www.ietf.org/rfc/rfc2281.txt

[HTTPShack] Pankaj Chowdhty, "The Gibraltar hack: Anatomy of a break−in", *PC Week Labs*, October 1999, http://www.zdnet.com/eweek/stories/general/0,11011,2350744,00.html

[Hunt] Pavel Krauz, "Hunt Project", http://lin.fsid.cvut.cz/~kra/index.html

[Hunt−FW] Ray Hunt, "Internet/Intranet Firewall Security − policy, architecture and transaction services", *Computer Communications, Elsevier, U.K., September 1998, Vol 21, No 13*

[ICMP] Jon Postel (Ed) "Internet Control Message Protocol", *RFC 792*, September 1981, http://www.ietf.org/rfc/rfc0792.txt

[IISWorm] CERT "sadmind/IIS Worm", *CERT Advisory CA−2001−11*, May 2001, http://www.cert.org/advisories/CA−2001−11.html

[IP] Jon Postel (Ed) "Internet Protocol", *RFC 791*, September 1981, http://www.ietf.org/rfc/rfc0791.txt

[IPChains] Rusty Russel, "Linux IP Firewalling Chains", October 2000, http://netfilter.samba.org/ipchains/

[IPSec] IETF IPSec Working Group, "IP Security Protocol (ipsec)", May 2001, http://www.ietf.org/html.charters/ipsec−charter.html

[IPStandards] J. Reynolds, R. Braden (Eds) "Internet Official Protocol Standards", *RFC 2700*, August 2000, http://www.ietf.org/rfc/rfc2700.txt

[IPv6] S. Deering, R. Hinden "Internet Protocol, Version 6 (IPv6) Specification", *RFC 2460*, December 1998, http://www.ietf.org/rfc/rfc2460.txt

[ISC] "Internet Domain Survey, January 2001", *Internet Software Consortium*, January 2001, http://www.isc.org/

[Joyal] Paul M. Joyal, "Industrial Espionage Today and Information Wars of Tomorrow", *19th National Information Systems Security Conference*, October 1996, http://csrc.nist.gov/nissc/1996/papers/NISSC96/joyal/industry.pdf

[KernHack] Paul "Rusty" Russel, "Unreliable Guide To Hacking The Linux Kernel", 2000, http://netfilter.samba.org/unreliable−guides/

[Kumar] Sandeep Kumar, "Classification and Detection of Computer Intrusions", *PhD Dissertation, Purdue University*, August 1995, ftp://coast.cs.purdue.edu/pub/COAST/papers/sandeep−kumar/kumar−intdet−phddis.ps

[Kyas] Othmar Kyas, "Internet Security − Risk Analysis, Strategies and Firewalls", *International Thomson Computer Press*, 1997, ISBN 1−85032−302−X

[Land] CERT, "IP Denial−of−Service Attacks", *CERT Advisory CA−1997−28*, December 1997, http://www.cert.org/advisories/CA−1997−28.html

[Law&Net] Law and the Net, "Infrastructure Threats from Cyber−Terrorists", March 1999, http://www.steptoe.com/WebDoc.NSF/Law+&+The+Net+All/Infrastructure+Threats+from+Cyber−Terrorists

[LHA] "High−Availability Linux Project", March 2001, http://linux−ha.org/

[Lion] Matt Fearnow, William Stearns, "Lion Worm", *SANS Institute Global Incident Analysis Centre*, April 2001, http://www.sans.org/y2k/lion.htm

[LKMPG] Ori Pomerantz, "Linux Kernel Module Programming Guide", *Linux Documentation Project*, April 1999, http://www.linuxhq.com/guides/LKMPG/mpg.html

[LocalDir] "Cisco LocalDirector", *Cisco Systems, Inc.*, http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml

[LSNAT] P. Srisuresh, D. Gan, "Load Sharing using IP Network Address Translation (LSNAT)", *RFC 2391*, August 1998, http://www.ietf.org/rfc/rfc2391.txt

[LVS] "Linux Virtual Server for Scalable Network Services", *Ottawa Linux Symposium 2000*, July 2000, http://www.linuxvirtualserver.org/Documents.html

[MiMattack] Bhavin Bharat Bhansali, "Man−In−the−Middle Attack − A Brief", *SANS*, February 2001, http://www.sans.org/infosecFAQ/threats/middle.htm

[MNLB] "High Availability Web Services", *Cisco whitepaper, Appendix A: Detailed Information about MNLB*, http://www.cisco.com/warp/public/cc/pd/ibsw/mulb/tech/mnlb_wp.htm

[MON] Jim Trocki, "−MON− Service Monitoring Daemon", http://www.kernel.org/software/mon/

[Morris] R. Morris, "A Weakness in the 4.2BSD UNIX TCP/IP Software", *Computing Science Technical Report No 117, ATT Bell Laboratories*, 1985, ftp://research.att.com/dist/internet_security/117.ps.Z

[MR] Eric Anderson, Dave Patterson, Eric Brewer, "The Magicrouter, an Application of Fast Packet Interposing", *Submitted to the Second Symposium on Operating System Design and Implementation*, May 1996, http://www.cs.berkeley.edu/~eanders/projects/magicrouter/

[NAT] P. Srisuresh, K. Egevang "Traditional IP Network Address Translator (Traditional NAT)", *RFC 3022*, January 2001, http://www.ietf.org/rfc/rfc3022.txt

[NCSA] E. Katz, M. Butler, R. McGrath, "A scalable Web server: the NCSA Prototype" *Computer Networks and ISDN Systems, 27, pp. 155−164, September 1994*, May 1994, http://archive.ncsa.uiuc.edu/InformationServers/Conferences/CERNwww94/www94.ncsa.ps

[Netcraft] "The Netcraft Web Server Survey", *Netcraft*, April 2001, http://www.netcraft.com/survey/

[NetDisp] Chris Gage, "IBM Network Dispatcher Version 3.0 − Scalability, Availability and Load−Balancing for TCP/IP applications", *IBM Whitepaper*, http://www−4.ibm.com/software/network/dispatcher/library/

[Netfilter] Rusty Russell, Marc Boucher, James Morris, Harald Welte "The Netfilter Project: Packet Mangling for Linux 2.3+", http://netfilter.samba.org/

[Netsizer] "Netsizer: Internet Growth Forecasting Tool", *Telcordia Technologies*, May 2001, http://www.netsizer.com/

[NFMod] Paul "Rusty" Russel, "Writing a Module for netfilter", *Linux Magazine*, June 2000, http://www.linux−mag.com/http://www.linux−mag.com/2000−06/gear_01.html

[NIDES] Debra Anderson, Thane Frivold, Alfonso Valdes, "Next−generation Intrusion Detection Expert System (NIDES): A summary", *SRI−CSL−95−07*, May 1995, http://www.sdl.sri.com/nides/reports/4sri.pdf

[nmap] Fyodor, "Nmap − The Network Mapper", http://www.insecure.org/nmap/index.html

[NTP] David L. Mills "Network Time Protocol (Version 3) Specification, Implementation", *RFC 1305*, March 1992, http://www.ietf.org/rfc/rfc1305.txt

[ONE−IP] Om P. Damani, P. Emerald Chung, Yennun Huang, Chandra Kintala, Yi−Min Wang, "ONE−IP: Techniques for Hosting a Service on a Cluster of Machines", *Proc. the Sixth Int. World Wide Web Conference*, April 1997, http://www.scope.gmd.de/info/www6/technical/paper196/paper196.html

[OPSEC] "Open Platform for Security", *Check Point Software Technologies*, 2001, http://www.checkpoint.com/opsec/architect.htm

[OSID] Fyodor, "Remote OS detection via TCP Stack Fingerprinting", October 1998, http://www.insecure.org/nmap/nmap−fingerprinting−article.txt

[PassiveID] Lance Spitzner, "Know Your Enemy: Passive Fingerprinting", *Honeypot Project*, May 2000, http://project.honeynet.org/papers/finger/

[PersFW] Zone Labs, "ZoneAlarm", http://www.zonelabs.com/products/za/moreinfo.html

[PingOfDeath] Malachi Kenney "Ping of Death", *Insecure.org*, October 1996, http://www.insecure.org/sploits/ping−o−death.html

[Piranha] Mike Wangsmo, "White Paper: Piranha − Load−balanced Web and FTP Clusters", RedHat Inc. Whitepaper, http://www.redhat.com/support/wpapers/piranha/

[Pix] "Cisco's PIX Firewall and Stateful Firewall Security", *Cisco Systems Inc.*, June 2000, http://www.cisco.com/warp/public/cc/pd/fw/sqfw500/tech/nat_wp.htm

[PktFilter] D. Brent Chapman "Network (In)Security Through IP Packet Filtering" *3rd USENIX Security Symposium*, September 1992, http://www.greatcircle.com/pkt_filtering.html

[PPTP] "Understanding PPTP (Windows NT 4.0)", *Microsoft Corporation*, January 2000, http://www.microsoft.com/technet/winnt/winntas/technote/pptpudst.asp

[Prices] NetworkFusion Magazine, "Load balancer server query results", http://www.nwfusion.com/bg/load/loadresult.jsp?_tablename=load

[PrivAddr] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear "Address Allocation for Private Internets", *RFC* 1918, February 1996, http://www.ietf.org/rfc/rfc1918.txt

[ProxyRFC] M. Chatel, "Classical versus Transparent IP Proxies", *RFC 1919*, March 1996, http://www.ietf.org/rfc/rfc1919.txt

[Ptachek] Thomas H. Ptacek, Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection", *Secure Networks Inc*, January 1998, http://www.securityfocus.com/data/library/ids.ps

[RAIN] Vasken Bohossian, Charles C. Fan, Paul S. LeMahieu, Marc D. Riedel, Lihao Xu, Jehoshua Bruck, "Computing in the RAIN: A Reliable Array of Independent Nodes", *Caltech Technical Report*, 1998, http://paradise.caltech.edu/papers/etr029.ps

[Ramen] CERT, "Widespread Compromises via "ramen" Toolkit", *CERT Incident Note IN−2001−01*, January 2001, http://www.cert.org/incident_notes/IN−2001−01.html

[Redir] Yuri Volobuev "Playing redir games with ARP and ICMP", http://staff.washington.edu/dittrich/papers/arp_fun.txt

[Requirements] R. Braden (Ed) "Requirements for Internet Hosts − Communication Layers", *RFC 1122*, October 1989, http://www.ietf.org/rfc/rfc1122.txt

[RProxy] "Netscape Proxy Server Administrator's Guide, Version 3.5 for Unix (Chapter 7: Reverse Proxy)", *Netscape*, http://developer.netscape.com/docs/manuals/proxy/adminux/revpxy.htm

[RRDNS] T. Brisco, "DNS Support for load−balancing", *RFC1794*, April 1995, http://www.ietf.org/rfc/rfc1794.txt

[Schuba] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, Diego Zamboni, "Analysis of a Denial of Service Attack on TCP", ftp://coast.cs.purdue.edu/pub/COAST/papers/christoph−schuba/schuba−krsul−kuhn−spaf−sundaram−zamboni−synkill.ps

[Securax] Securax "Ms Windows '95/'98/SE will crash upon parsing special cred path−strings refering to device drivers.", March 2000, http://securax.org/pers/scx−sa−01.txt

[SmCli] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, David Culler, "Using Smart Clients to Build Scalable Services", USENIX 1997 Annual Technical Conference, January 1997, http://www.usenix.org/publications/library/proceedings/ana97/full_papers/yoshikawa/yoshikawa_html/usenix97.html

[Smurf] CERT, "Smurf IP Denial−of−Service Attacks", *CERT Advisory CA−1998−01*, January 1998, http://www.cert.org/advisories/CA−1998−01.html

[SOCKS] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones "SOCKS Protocol Version 5", *RFC 1928*, March 1996, http://www.ietf.org/rfc/rfc1928.txt

[Sommer−1] Peter Sommer, "Computer Forensics: an introduction", 1997, http://virtualcity.co.uk/vcaforens.htm

[Sommer−2] Peter Sommer, "Intrusion Detection Systems as Evidence", *RAID98*, http://www.zurich.ibm.com/pub/oOther/RAID/Prog_RAID98/Full_Papers/Sommers_text.pdf

[Spafford] Eugene H. Spafford, "The Internet Worm Incident", Purdue Technical Report CSD−TR−933, Proc. of the 1989 European Software Engineering Conference, September 1991, ftp://coast.cs.purdue.edu/doc/morris_worm/spaf−Iworm−paper−ESEC.ps.Z

[Stallings] William Stallings "Data and Computer Communications", *Prentice−Hall Inc. ISBN 0−13−086388−2*, 6th Edition, 2000

[Stat] James T. McClave, Terry Sincich, "Statistics", *Prentice−Hall*, 2000, ISBN 0−13−022329−8

[Stevens−1] W. Richard Stevens "Unix Network Programming: Volume 1", *Prentice−Hall ISBN 0−13−490012−X*, 2nd Edition, September 1997, http://www.kohala.com/start/unpv12e.html

[Stevens−2] W. Richard Stevens "TCP/IP Illustrated, Volume 1: The Protocols", *Addison−Wesley ISBN 0−201−63346−9 (v.1)*, 1994

[Stonebeat] "Stonesoft White Papers", *Stonesoft Corp.*, http://www.stonesoft.com/document/203.html

[Subnet] J. Mogul, J. Postel "Internet Standard Subnetting Procedure", *RFC 950*, August 1985, http://www.ietf.org/rfc/rfc0950.txt

[SWEB] D.Andresen, T.Yang, V.Holmedahl, O.Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers ", *Proc. of 10th IEEE International Symp. on Parallel Processing (IPPS'96), pp. 850−856*. April, 1996, http://www.cs.ucsb.edu/~tyang/papers/IPPS96_sweb.ps

[SYNflood] CERT, "TCP SYN Flooding and IP Spoofing Attacks", *CERT® Advisory CA−1996−21*, April 1996, http://www.cert.org/advisories/CA−1996−21.html

[TCP] Jon Postel (Ed) "Transmission Control Protocol", *RFC 793*, September 1981, http://www.ietf.org/rfc/rfc0793.txt

[TCPHijack] L. Joncheray, "Simple Active Attack Against TCP",.*Proceedings, 5th USENIX UNIX Security Symposium*, June 1995, http://www.usenix.com/publications/library/proceedings/security95/full_papers/joncheray.txt

[TCP−HP] V. Jacobson, R. Braden, D. Borman "TCP Extensions for High Performance", *RFC 1323*, May 1992, http://www.ietf.org/rfc/rfc1323.txt

[TCPISN] CERT, "Statistical Weaknesses in TCP/IP Initial Sequence Numbers", *CERT Advisory CA−2001−09*, May 2001, http://www.cert.org/advisories/CA−2001−09.html

[TCPOpt] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, Larry L. Peterson, "Optimizing TCP forwarder performance", *IEEE/ACM Transactions on Networking vol. 8 no. 2 pp. 146−157*, 2000, ftp://ftp.cs.arizona.edu/reports/1998/TR98−01.ps

[TelnetHijack] Ed Norris, "Analysis of a Telnet Session Hijack via Spoofed MAC Addresses and Session Resynchronization", *SANS*, March 2001, http://www.sans.org/infosecFAQ/threats/hijack.htm

[ToS/DS] K. Nichols, S. Blake, F. Baker, D. Black "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", *RFC 2474*, December 1998, http://www.ietf.org/rfc/rfc2474.txt

[UDP] Jon Postel (Ed) "User Datagram Protocol", *RFC 768*, August 1980, http://www.ietf.org/rfc/rfc0768.txt

[Walrus] Yair Amir, David Shaw, "WALRUS − a Low Latency, High Throughput Web Service Using Internet−wide Replication", *19th IEEE ICDCS Workshop on Electronic Commerce and Web−Based Applications, pp. 31−40, Austin*, May 1999, http://www.cs.jhu.edu/~yairamir/

[WGHA] "WatchGuard High Availability", *WatchGuard Technologies, Inc.*, 2001, http://www.watchguard.com/products/highavail.asp

[whisker] Rainforest Puppy, "Whisker information, scripts, and updates", May 2001, http://www.wiretrip.net/rfp/p/doc.asp?id=21

[Whois] K. Harrenstien, M. Stahl, E. Feinler, "NICNAME/WHOIS", *RFC 954*, October 1985, http://www.ietf.org/rfc/rfc0954.txt

[WinNuke] Computer Incident Advisory Centre, "H−57: Windows NT/95 Out of Band Data Exploit", May 1997, http://ciac.llnl.gov/ciac/bulletins/h−57.shtml

[WWWSeqFaq] World Wide Web Consortium, "The World Wide Web Security FAQ", April 2000, http://www.w3.org/Security/Faq/wwwsf4.html

[Zwicky] Elizabeth D. Zwicky, Simon Cooper, D. Brent Chapman "Building Internet Firewalls", *O'Reilly ISBN 1−56592−871−7*, 2nd Edition, June 2000

15. Appendix 1: A time−bounded, reliable message protocol

When transferring a rule token between nodes, the GLOB system has a number of problems:

1. A message must somehow make its way from the source node (node A), to the destination node (node B).
2. The protocols available to this must be some derivative of IP − TCP or UDP based being most likely.
3. The rule token may be present at no more than one node at a time − ensuring that a client does not receive contradictory replies to a request.
4. The rule token can be disabled while in transit, but this transit time must be bounded.
5. Accidentally duplicating or losing a token is highly undesirable.
6. The message passing may fail, within the bounds set above, as long as both A and B agree on where the token is at the end of communication.
7. The message passing protocol should not overload any network links between A and B − since it will typically be used when that link is already heavily loaded.

One possible approach is a simple, message−acknowledgement system. TCP [TCP], for example, uses a similar scheme − once the initial handshake is complete. After sending a data segment, TCP sets a timer. Unless the segment is acknowledged in time, TCP forces a retransmission (until a second timer expires, when the connection fails).

Such a system does not comply with these requirements. Consider the following scenario:

1 A sends a rule token to B. A still considers itself to hold the token, however.

2 B may or may not receive the message.

2.1 If B does not receive the token, A does not receive any response and enables the rule locally. (Transfer has failed).

2.2 If B does receive the token, it generates and sends an acknowledgement (ACK). B believes it now has the token.

3 A may or may not receive B’s ACK message.

3.1 If A receives it, A knows B holds the token, and releases it. Transfer has succeeded.

3.2 If A does not receive the ACK, it will eventually time out and enable the

rule locally.

In both cases 2.1 and 3.2, the transfer has failed. In both cases, A believes that it continues to hold the token − but in 3.2, B believes that it holds the token. In theory, this problem could be resolved by A sending an ACK on B's ACK message − which, in turn, becomes subject to this problem.

Therefore, a simple single−ACK based system cannot meet criterion 6 above − A and B does not agree on the location of the token. Since adding additional acknowledgement steps do not increase the number of packet failures needed to break the system, this scheme does not allow minimisation of the failure probability.

15.1. Mark 2 − the better way

The problem with the above lies in the fact that a sender can never be sure that the last message arrived, unless get a reply is received. However, by assuming that successive messages have an independent chance of failing, an additional improvement is possible: multiple copies of the same message must all fail, in order for the message to fail.

In order to develop this concept, we start by defining some concepts:

1. Any given message has a probability of failing no greater than *p*, which is assumed to be smaller than 0.5 (generally much smaller).
2. Messages take no more than *t* seconds to travel from A to B, or vice versa. This is a relatively stable number in the case being considered, where A and B are on the same network segment.
3. A and B have externally synchronised clocks, using a mechanism such as NTP [NTP].

A sends up to *n* rule tokens to B, at intervals large enough for a return trip. Every time B receives a token, it sends an ACK. Each token includes a time value of the start of the sending process. When A receives an ACK, it stops sending rule tokens. If B receives another request, it knows that the previous ACK messages were lost. If A has not had an ACK after *n* sends, it must assume that the connection has failed − it sends a cancel message (retrying this a number of times).

For *n=1*, this is a simple acknowledgement scheme. If *n=2*, the scheme fails only if at least two messages get lost − both ACK messages, or the first ACK and the second token. Consider the possible outcomes:

1. A holds the token, and sends a token message to B.

1.1 The first token message is lost (*p* chance), and the ystem waits until retransmit time.

1.2 The first token message is received. (*1−p*) The next state is State 2.

2. B sends an ACK message.

2.1 The ACK is lost, *(1−p)(p)* the system waits for retransmit.

2.2 A receives an ACK. *(1−p)(1−p)* The message has been received ok.

After one message iteration, there is a *(1−p)(1−p)* chance of success, and a *p(2−p)* chance of failure. If further iterations occur, this chance is compounded. If no further iterations occur, there is a *p* chance of a correct failure, and a *(1−p)(p)* chance of a rule duplication. Alternatively, by assuming that a non−reply is a success, there is a *p* chance of rule loss (which is problematic), and a *(1−p)(p)* chance of acceptable failure.

For every further iteration, these probabilities are multiplied by the compound of the previous iteration. To illustrate, consider the case where *p=0.1*, in Table 13.

N	***P(success)***	***P(loss)***	***P(fail)***	***Notes***
*1	0.9	0.1	0	Send & forget
*2	0.81	0.09	0.1	Simple ACK
1	0.81	0.1	0.09	
2	0.9639	0.019	0.0171	
3	0.9931	0.0036	0.0032	

Table 13. Evaluation of the transfer schemes for different numbers of messages for p=0.1

As can be seen, the probability of failure decreases by an order of *p(2−p)* for every increase in *n*. The maximal value of *n* depends on the ratio of *T/t* − the system must allow time for replies to arrive before timing out.

A problem with this scheme is the fact that *P(loss)* is greater than *P(fail)* − an artefact of the preference for rule loss over duplication. One method of addressing this lies in including failure notification. When a failure is indicated, A retains the rule token. At this point, B might have a duplicate token − leading to overlapping task loads.[75] A then proceeds to enable the rule locally, and sends regular failure messages to B. When B receives such a message, it must discard the held rule immediately.

Rule duplication can only happen when B has acknowledged a rule token, but A has not received the acknowledgement. Using the first configuration, B could

75 This would have no effect on connections that are simply routed − but produces multiple, possibly contradictory, responses to requests that require state−dependent handling.

immediately enable the rule token − once acknowledged, A will not attempt to reinstate the rule. In the second configuration, B could wait to allow failure messages from A to arrive, or could simply take the new rule into use. In the first case, *T* is prolonged without increasing *n*; in the second case an increased risk of duplicate responses occurs.

The latter approach is currently considered preferable, since the probability of communication failure is expected to be small.

16. Appendix 2: Proxy log during resumption

05/03 12:39:40 14300 1 Logging to /tmp/proxy.log with flags 0x1A
05/03 12:39:40 14300 10 Server mode (iterative)
Iterative mode allows the right instance to be interrupted − does HTTP requests in a single process, in order
05/03 12:39:49 14300 8 Request 132.181.8.5:51191 −> 132.181.8.5:8001 (proxy)
Direct connection to the proxy, so it ducks behind the University proxy.
05/03 12:39:49 14300 10 Connecting :0 −> 132.181.3.233:80
05/03 12:39:49 14300 8 Connected 132.181.8.5:51192 −> cantud.canterbury.ac.nz:80
05/03 12:39:49 14300 10 Request:
GET http://mysite.ciaoweb.it/edgenis/cashel1.htm HTTP/1.0
Referer: http://mysite.ciaoweb.it/edgenis/ireland_home.htm
Connection: close
User−Agent: Mozilla/4.75 [en] (X11; U; Linux 2.4.0−test9 i686)
Pragma: no−cache
Host: mysite.ciaoweb.it
Accept: image/gif, image/x−xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept−Language: en
Accept−Charset: iso−8859−1,*,utf−8
Cookie: SITESERVER=ID=2add8ad3b8e1e8eeeffd23319091e552
Proxy−Connection: close

05/03 12:39:49 14300 10 Output header (Client):
GET http://mysite.ciaoweb.it/edgenis/cashel1.htm HTTP/1.0
Referer: http://mysite.ciaoweb.it/edgenis/ireland_home.htm
Connection: close
User−Agent: Mozilla/4.75 [en] (X11; U; Linux 2.4.0−test9 i686)
Pragma: no−cache
Host: mysite.ciaoweb.it
Accept: image/gif, image/x−xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept−Language: en
Accept−Charset: iso−8859−1,*,utf−8
Cookie: SITESERVER=ID=2add8ad3b8e1e8eeeffd23319091e552
Proxy−Connection: close
Via: httpproxy/0.2 (KRN)

05/03 12:39:49 14300 10 Pending: GET http://mysite.ciaoweb.it/edgenis/cashel1.htm HTTP/1.0
05/03 12:39:57 14300 10 Request:
HTTP/1.1 200 OK
Date: Thu, 03 May 2001 00:42:06 GMT
Age: 0
Server: Microsoft−IIS/4.0
Content−Type: text/html
Accept−Ranges: bytes
Last−Modified: Sun, 22 Oct 2000 21:38:31 GMT
Content−Length: 593
Etag: "966f316c703cc01:960b"
Via: 1.1 NetCache (NetCache 4.1R5D2)

05/03 12:39:57 14300 10 Output header (Server):
HTTP/1.1 200 OK
Date: Thu, 03 May 2001 00:42:06 GMT
Age: 0
Server: Microsoft−IIS/4.0
Content−Type: text/html
Accept−Ranges: bytes

Last−Modified: Sun, 22 Oct 2000 21:38:31 GMT
Content−Length: 593
Etag: "966f316c703cc01:960b"
Via: 1.1 NetCache (NetCache 4.1R5D2)
Via: httpproxy/0.2 (KRN)
Request: GET http://mysite.ciaoweb.it/edgenis/cashel1.htm HTTP/1.0

05/03 12:39:57 14300 10 Expecting block (593)
05/03 12:39:57 14300 10 Server close
05/03 12:39:57 14300 10 Server close
05/03 12:39:57 14300 8 Transfer 533/507 > | < 960/866 characters (total 533, 960) (block 593/593)
05/03 12:39:57 14300 8 Closing connection to server
Transfer of uninterrupted version done
05/03 12:39:57 14300 8 Request 132.181.8.5:51194 −> 132.181.8.5:8001 (proxy)
05/03 12:39:57 14300 10 Connecting :0 −> 132.181.3.233:80
05/03 12:39:57 14300 8 Connected 132.181.8.5:51195 −> cantud.canterbury.ac.nz:80
05/03 12:39:57 14300 10 Request:
GET http://mysite.ciaoweb.it/edgenis/thumbnails/cashel1.jpg HTTP/1.0

Referer: http://mysite.ciaoweb.it/edgenis/cashel1.htm
Connection: close
User−Agent: Mozilla/4.75 [en] (X11; U; Linux 2.4.0−test9 i686)
Pragma: no−cache
Host: mysite.ciaoweb.it
Accept: image/gif, image/x−xbitmap, image/jpeg, image/pjpeg, image/png
Accept−Language: en
Accept−Charset: iso−8859−1,*,utf−8
Cookie: SITESERVER=ID=2add8ad3b8e1e8eeeffd23319091e552
Proxy−Connection: close

05/03 12:39:57 14300 10 Output header (Client):
GET http://mysite.ciaoweb.it/edgenis/thumbnails/cashel1.jpg HTTP/1.0
Referer: http://mysite.ciaoweb.it/edgenis/cashel1.htm
Connection: close
User−Agent: Mozilla/4.75 [en] (X11; U; Linux 2.4.0−test9 i686)
Pragma: no−cache
Host: mysite.ciaoweb.it
Accept: image/gif, image/x−xbitmap, image/jpeg, image/pjpeg, image/png
Accept−Language: en
Accept−Charset: iso−8859−1,*,utf−8
Cookie: SITESERVER=ID=2add8ad3b8e1e8eeeffd23319091e552
Proxy−Connection: close
Via: httpproxy/0.2 (KRN)

05/03 12:39:57 14300 10 Pending: GET
http://mysite.ciaoweb.it/edgenis/thumbnails/cashel1.jpg HTTP/1.0
05/03 12:40:04 14300 10 Request:
HTTP/1.1 200 OK
Date: Thu, 03 May 2001 00:42:14 GMT
Age: 0
Server: Microsoft−IIS/4.0
Content−Type: image/jpeg
Accept−Ranges: bytes
Last−Modified: Sun, 22 Oct 2000 22:09:52 GMT
Content−Length: 209097
Etag: "9b6aecd743cc01:a1fa"
Via: 1.1 NetCache (NetCache 4.1R5D2)

05/03 12:40:04 14300 10 Output header (Server):
HTTP/1.1 200 OK
Date: Thu, 03 May 2001 00:42:14 GMT
Age: 0
Server: Microsoft−IIS/4.0
Content−Type: image/jpeg
Accept−Ranges: bytes
Last−Modified: Sun, 22 Oct 2000 22:09:52 GMT
Content−Length: 209097
Etag: "9b6aecd743cc01:a1fa"
Via: 1.1 NetCache (NetCache 4.1R5D2)
Via: httpproxy/0.2 (KRN)
Request: GET http://mysite.ciaoweb.it/edgenis/thumbnails/cashel1.jpg HTTP/1.0

05/03 12:40:04 14300 10 Expecting block (209097)
Loading image...
05/03 12:40:06 14300 20 SIGHUP: Backing down...
Interrupted: old proxy terminates, new proxy process started
05/03 12:40:07 14306 1 Logging to /tmp/proxy.log with flags 0x1A
05/03 12:40:07 14306 1 Resuming from /var/run/proxy.pst.14300
05/03 12:40:08 14306 40 Freed 17568 proxy state for 14306
05/03 12:40:08 14306 40 Allocated 17568 proxy state for 14306
Loaded up old proxy process state
05/03 12:40:08 14306 8 132.181.8.5:51194−>132.181.8.5:8001 <=>
132.181.8.5:51195−>132.181.3.233:80
05/03 12:40:13 14306 4 Reaping 14308
05/03 12:40:13 14306 4 Reaping 14307
Slave processes that help with opening hijacked connections died
05/03 12:40:16 14306 10 Server close
05/03 12:40:16 14306 8 Transfer 0/0 > | < 146741/146741 characters (total 534, 172798)
(block 172417/209097) (1 resumes)
Page load completes: the new proxy loaded the last 146 kB of the file, with about 35 kB lost during the interruption
05/03 12:40:16 14306 40 Freed 17568 proxy state for 14306

17. Appendix 3: Source Code

During the course of this thesis, a number of programs and scripts were developed. This appendix provides an overview of these files, with printed versions of the complete files[76] appended to the thesis. All of the source code and fragments included in this document should be considered to be subject to the Gnu General Public License [GPL], except for the fragments based on code from [Stevens−1]. Refer to [NFMod], [LKMPG], and [KernHack] for details on Linux kernel and Netfilter programming.

17.1. Proxy source code

File name	***Function***
proxy.h	Header file and source code for connecting the proxy framework and protocol−specific proxies.
proxy.c	The basic source code for the protocol−neutral TCP proxy and proxy framework.
httpproxy.c	The source code for the HTTP proxy. Links to the proxy framework in *proxy.c*.
stun.c	Test program supporting proxy resume testing. Automates the replacement of an active proxy with a resumed proxy process.

Table 14. Core proxy source code files

17.2. Library and general source code

Found in the *lib* directory, these files contain general−use and supporting functions. Only header files are included to reduce the length of this thesis − implementation files are available on request.

File name	***Function***
cbuffer.{h,c}	Circular buffer maintenance functions.
cmdline.{h,c}	Commandline options support functions for use with *getopt()*.
general.{h,c}	General−purpose support and process control functions.
log.{h,c}	Unified error logging library.
mdump.{h,c}	Memory−mapped file allocation and loading functions.
netlib.{h,c}	Network support functions: connection maintenance, addressing, network socket I/O and miscellanious network−related functions.
read_fd.c *write_fd.c*	File descriptor passing functions based on page 387 of [Stevens−1], used in the proxy resume framework before *prism* was implemented. Uses Unix domain sockets to pass open file descriptors between processes.
writefd.{h,c}	Prototypes for *read_fd.c* and *write_fd.c*, and contains other Unix domain sockets−based functions.

76 In the source code, /** ... */ indicates items that may require further attention.

File name	Function
seqno.{h,c}	Sequence number extraction functions, for TCP connections. Requires kernel modifications that print out the current sequence numbers for connections as part of */proc/net/tcp*. See Section 8.1 for more details.
strargv.{h,c}	Translate a string into an *argv*−style array of 0−seperated substrings, used to extend commandline parameters with configuration file options.

Table 15. Supporting source code files

17.3. prism *and* prism *support source code*

The prism module, while technically part of the proxy framework, is unusual enough to warrant separate consideration. For more information on the *Netfilter* framework and interfaces, see [Netfilter] and [NFMod].

File name	Function
prism.h	Structure and function prototypes for communication between the *prism* kernel module and client applications.
prism.c	Kernel module for *prism*. Includes the functions used to support interaction via the */proc/prism* file, and the functions for *Netfilter* hooks that handle packet modifications.
lib/prismclient.c	Contains the *hijack_connection* function (see Section 8.4.1) supporting connection take−over or resumption.
intercept.c	Testing client for *prism* that places a simple snooping transparent proxy into an existing connection.
hijack.c	Testing client for *prism* that demonstrates connection hijacking.

Table 16. *prism* source code files

17.4. GLOB model source code

This section includes a selection of the scripts used to generate GLOB and other model statistics.

File name	Function
chainmodel.c	Model packet filtering structure, emulating the behaviour of the network interfaces and packet filter systems in a real GLOB implementation.
rule.h	Message formats between *model.c* and *chainmodel.c*.
ipt_lb.awk	BALANCE model packet scheduler.
lc_lb.awk	Least Connections model packet scheduler.
do_srr.sh	Simple Round Robin model scheduler and test script.
lbclient.{h,c}	*Lbmodel* core GLOB functions
model.c	*Lbmodel* functions specific to the model environment. Expected to be replaced for a network implementation of GLOB.

Table 17. GLOB source code files